La programmazione dello Z-8000

EDIZIONE ITALIANA

RICHARD MATEOSIAN

SYBEX

GRUPPO EDITORIALE JACKSON

La programmazione dello Z-8000

di
Richard
Mateosian

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

Hanno contribuito:
Copertina: Daniel Le Noury. **Progetto grafico:** Roger Gottlieb. **Composizione:** Diana J. Goodin. **Disegni tecnici:** j. truyillo smith. **Produzione:** Guy S. Orcutt.

AVVERTENZE

Si è cercato per quanto possibile, di fornire informazioni complete e rigorose. In ogni caso la Sybex non si assume alcuna responsabilità per il loro impiego; nemmeno al riguardo di infrazioni di brevetti e di altri diritti di terze parti che ne potrebbero derivare. I costruttori di apparecchiature non rilasciano alcuna autorizzazione su apparecchiature protette da brevetto o diritti di brevetto e si riservano la facoltà di cambiare, in qualunque momento, la disposizione circuitale senza alcun preavviso.

In particolare sono soggetti a frequente cambiamento le caratteristiche tecniche e i prezzi. I confronti e le valutazioni sono presentati solo per il loro valore educativo ed i loro principi informativi. Per le specifiche esatte si rimanda il lettore ai dati del costruttore.

© Copyright per l'edizione originale SYBEX Inc. 1980, 2020 Milvia Street – Berkeley, California 94704
© Copyright per l'edizione italiana SYBEX Inc. 1981
Tutti i diritti sono riservati – Nessuna parte di questo libro può essere riprodotta, posta in sistemi di archiviazione, trasmessa in qualsiasi forma o mezzo, elettronico, meccanico, fotocopiatura etc., senza l'autorizzazione scritta dall'editore.

Avvertenza:
Le disposizioni dei piedini dello Z8001 e dello Z8002 sono riportate a pag. 51 col permesso © della Zilog Inc.
«Z8001», «Z8002» e «Zilog» sono marchi registrati della Zilog Inc. Questi nomi nel libro sono sempre usati in forma complementare ai termini microprocessor o Chip CPU, anche quando questi ultimi non appaiono esplicitamente.
La Sybex non ha alcun legame con la Zilog Inc.

Fotocomposizione: Corponove s.n.c. – Bergamo
Stampa: Tipo Lito Ferrari Cesare & C. – Clusone (BG)

INDICE

Prefazione XII

Introduzione 1

I. Concetti base **5**
Che cosa è la programmazione? 5
Algoritmi 5
Diagrammi di flusso 8
Rappresentazione delle informazioni 8
Rappresentazione esadecimale 8
Complemento a due 18
Rappresentazione BCD 21
Rappresentazione in virgola mobile 23
Rappresentazione di caratteri alfanumerici 25

II. Architettura circuitale dello Z8000 **29**
Registri non specializzati 30
Stack 32
Segmentazione della memoria 34
Mappa della memoria 35
Spazi multipli di indirizzi 37
Modi Sistema e Normale 39
Lo Z8001 e lo Z8002 39
Stato della CPU 40
Trappole e interruzioni 42
Registri di controllo 47
Elaborazione distribuita 48
Il dispositivo CPU 50

III. Introduzione alle tecniche di programmazione **53**

IV. Parte I — Istruzioni dello Z8000 **77**
Operazione e controllo sui dati 78
Trasferimento dati 84
Posizionamento del puntatore 88
Trasferimento del Controllo 88
Ingresso/Uscita 90
Controllo del CPU 90
Operazioni di blocco 91

Sincronizzazione Multi-Micro 94
Codici condizione 95
Formato delle istruzioni 97
Descrizione delle istruzioni 102
Indici 105
Come utilizzare le Descrizioni delle Istruzioni 105
Alcune considerazioni di progetto 117

Parte II
Indice delle Descrizioni delle Istruzioni (alfabetico) 118
Indice delle Descrizioni delle Istruzioni (mediante il valore numerico del codice operativo) 120
Descrizione delle 45 istruzioni 122

V. Modi di indirizzamento dello Z8000 181

VI. Tecniche per la gestione dell'Ingresso/Uscita 189
Trasmissione seriale sincrona di bit di dati 192
Trasmissione parallela sincrona dei dati 195
Larghezza della banda di trasmissione 197
Esempio di visualizzazione a sette segmenti 198
Catalogatori di I/O 201
Un esempio 204

VII. Componenti periferici dello Z8000 217
Istruzioni speciali di I/O 217
Unità di gestione della memoria (MMU) 218
La famiglia di componenti dello Z8000 220
I/O seriale 220
I/O parallelo 221
Circuiti contatori/temporizzatori 221
Alcune caratteristiche non pianificate 221

VIII. Esempi di programmi di utilità 223
Inizializzazione degli I/O 223
Routine di gestione del buffer ad anello 229
La routine di gestione della linea di memorizzazione caratteri 231
La routine di conversione 231
Programma di interazione con il terminale 235
Decodifica dell'ingresso 240
Formattazione dell'uscita 244

Controllo cursore 248
Tabelle ai bit 252

IX. Tecniche avanzate di programmazione 255
Programmi condivisibili e gestione della memoria 255
Time-sharing 257
Gestione dello stack 267
Un meccanismo di invio del SC 272
Inizializzazione del sistema 275

X. L'ambiente di sviluppo dei programmi 282
Editori di testo 285
Assemblatori 286
Strumenti per il debug 290
Hardware per lo sviluppo di programmi 292

Appendici 295
A Risposte agli esercizi selezionati 295
B Insieme dei caratteri ASCII 298
C Codifica dei campi delle istruzioni 299
D Formati del registro di controllo 301
E Struttura dell'area di stato del programma 302

FIGURE

Capitolo I

1.1 Moltiplicazione di due numeri 6
1.2 Algoritmo della moltiplicazione 6
1.3 Diagramma di flusso dell'algoritmo della moltiplicazione ... 9
1.4 Configurazioni e raggruppamento di bit 10
1.5 Nomi delle configurazioni esadecimali dei bit 11
1.6 Relazione tra rappresentazione binaria ed esadecimale 13
1.7 Uso di n bit per distinguere 2^n elementi 14
1.8 Algoritmo per la conversione da decimale a esadecimale .. 15
1.9 Conversione di 299_{10} in $12B_{16}$ 16
1.10 Tabella di addizione e moltiplicazione esadecimale 17
1.11 Moltiplicazione esadecimale 18
1.12 Una parola di 16 bit interpretata come odometro esadecimale 19
1.13 Rappresentazione BCD 22
1.14 Rappresentazione in virgola mobile di 0,34375 25
1.15 Insieme dei caratteri ASCII 27
1.16 Rappresentazione di un testo ASCII 28

Capitolo II

2.0 Alcuni usi dei registri 30
2.1 Gerarchia dei registri non specializzati dello Z8000 32
2.2 Stack 33
2.3 Formato del registro per gli indirizzi segmentati 35
2.4 Alcune caratteristiche della mappatura di memoria 36
2.5 Calcolo dell'indirizzo mappato 38
2.6 Identificazione del significato dei bit nella parola degli indicatori di controllo (FCW) 40
2.7 Stato della CPU per interruzioni e LDPS 42
2.8 Stack dopo un'interruzione 44
2.9 Stato del programma per le interruzioni e le trappole 46
2.10 Localizzazione dei campi del registro REFRESH 48
2.11 Denominazione dei piedini dello Z8001 e Z8002 51
2.12 Codifica delle uscite di stato dello Z8000 51

Capitolo III

3.1 Algoritmo E 55

3.2 Programma dell'algoritmo E 58
3.3 Uso del contatore di locazione in fase di assemblaggio ... 60
3.4 Uso dello stack per le istruzioni CALL e RET 68
3.6 Confronto delle due versioni di LENGTH 71
3.7 Subroutine LENGTH per il programma di crittografia 72
3.8 Subroutine MOD23 per il programma di crittografia 73
3.9 Versione a subroutine del programma di crittografia 75

Capitolo IV

4.1 Percorso dati per istruzioni aritmetiche, logiche e spostamenti/rotazioni 79
4.2 Tabella della verità per operazioni logiche 80
4.3 Alcuni spostamenti e rotazioni 81
4.4 La circolazione di digit realizzata con R($\frac{L}{R}$)DE 82
4.5 Percorso dati per le istruzioni che modificano il contatore/puntatore 82
4.6 Percorso dati per le istruzioni di test 83
4.7 (a) Percorso dati per l'istruzione di trasferimento dati e posizionamento del puntatore – alcuni trasferimenti di costanti 85
4.7 (b) Percorso dati per le istruzioni di trasferimento dati e posizionamento puntatore – scambio e caricamento 86
4.7 (c) Percorso dati per le istruzioni di trasferimento dati e posizionamento puntatore – estensione del segno 87
4.7 (d) Percorso dati per le istruzioni di trasferimento dati e posizionamento puntatore – trasferimenti su stack 87
4.8 Percorso dati per le istruzioni di controllo del CPU 92
4.9 Registri e sequenza di funzionamento per le versioni a blocco di CP, LD, I/O 93
4.10 Funzionamento delle istruzioni di conversione di test 94
4.11 Categorie funzionali delle istruzioni 96
4.12 Codici condizione 97
4.13 Prima approssimazione del formato delle istruzioni dello Z8000 98
4.14 Le quattro istruzioni speciali concepite per un'elevata densità di codifica 99
4.15 Una tipica istruzione a due operandi 100
4.16 Codifica del modo di indirizzamento mediante modo e registro 102
4.17 Condizione rappresentata dai bit di FLAGS 104
4.18 Tempo (cicli) addizionale per l'indirizzamento segmentato 105

4.19 Versione assemblata di LENGTH 114
4.20 Programma misterioso 116

Capitolo V

5.1 Modi di indirizzamento dello Z8000 182
5.2 Formati degli indirizzi usati nelle istruzioni 183
5.3 Indirizzi nelle istruzioni 184
5.4 Esempi di argomenti immediati 185
5.5 Esempi di modo di indirizzamento su base e indirizzo su base 186
5.6 Interpretazione dei campi di registro delle istruzioni 187

Capitolo VI

6.1 Algoritmo per la lettura dei caratteri provenienti da TTY 194
6.2 Subroutine per la lettura dei caratteri da TTY 195
6.3 Versione di INCH per trasmissione parallela 196
6.4 Versione dell'uscita equivalente ad INCH 197
6.5 Velocità di trasferimento parallelo dello Z8000 198
6.6 Codice a sette segmenti 199
6.7 Programma per la visualizzazione mediante LED 201
6.8 Tipica sequenza di polling 202
6.9 Pseudocodice della routine di servizio dell'uscita per terminale 205
6.10 Pseudocodice della routine di servizio di ingresso da terminale 298
6.11 Routine di commutazione del contesto per l'I/O da terminale 210
6.12 Routine di servizio di uscita per terminale (escluso eco) . 212
6.13 Routine di servizio di uscita per terminale (parte eco) ... 213
6.14 Routine di servizio per ingresso da terminale 214

Capitolo VIII

8.1 Sequenza degli eventi di un ingresso da tastiera 225
8.2 Inizializzatore dell'I/O per il terminale 227
8.3 Definizioni e formati dei buffer ad anello 230
8.4 Routine di gestione del buffer ad anello 233
8.5 Funzionamento di un buffer ad anello 233
8.6 BACKUP e ADLINE 234

8.7 La routine TRAN per la ricerca associativa 235
8.8 Definizioni e formati delle routine di LINE 237
8.9 Routine LINE 238
8.10 ASK e SAY 239
8.11 Routine campione per la decodifica della linea di ingresso 243
8.12 Esempio di codice per l'elaborazione dell'ingresso 243
8.13 Identificatore di comando 245
8.14 Alcune routine per la formazione della linea 246
8.15 Aggiunte alle routine di linea 249
8.16 Programmi di controllo cursore 250
8.17 Sequenza di chiamata e definizione per un programma di tabella a bit 251
8.18 Routine di gestione della tabella a bit 252
8.19 Specificazione dell'entrata e uscita della routine del programma nella tabella a bit 253

Capitolo IX

9.1 Una possibile struttura dei dati dell'utente 258
9.2 Una struttura migliore 259
9.3 Revisione della routine di inizializzazione TYIO 263
9.4 Struttura dei dati dell'utente 264
9.5 Blocchi di contesto del terminale 265
9.6 Una semplice lista di circolazione per uno scambiatore per time-sharing 266
9.7 Salvataggio dello stato di attesa utente di I/O 268
9.8 Separazione degli stack di chiamata e di memorizzazione 269
9.9 Argomenti inviati indietro tramite lo stack di memorizzazione 270
9.10 Flusso di controllo quando si usa un programma di entrata del sistema 271
9.11 Routine di entrata del sistema per la gestione dello stack 272
9.12 Gestore minimo per SC 275
9.13 Area di stato di programma 277
9.14 Un'implementazione esterna della trappola per «indirizzi dispari di parola» 279
9.15 Codifica dell'inizializzazione del sistema 280

Capitolo X

10.1 Una possibile configurazione per lo sviluppo dei programmi 293

PREFAZIONE

I microprocessori si sono sviluppati rapidamente negli anni '70 e verso la fine di questo periodo la tecnologia dell'integrazione su larga scala ha raggiunto un livello tale da poter supportare la più avanzata architettura dei minicalcolatori. I microprocessori a sedici-bit lanciati nel 1980 non sono solamente delle pietre miliari — essi sono dei completi calcolatori che eserciteranno una notevole influenza sulle applicazioni degli anni a venire. Questi nuovi microprocessori dovrebbero essere studiati da tutte le persone che nel 1980 lavorino o progettino dei calcolatori.

Per questa ragione considero personalmente che un libro sullo Z8000 sia una realizzazione importante ed un positivo impiego di tempo ed energia. Quest'idea condivisa da Rodnay Zaks, presidente della Sybex, e da me è stata supportata in modo eccellente ad ogni stadio del mio lavoro; dal disegno della copertina del libro alla scrupolosa esecuzione di ogni mio fantasioso volo letterario.

Alla Sybex, tutti mi hanno aiutato e incoraggiato, ma ci sono varie persone che hanno contribuito a questo libro in modo particolarmente valido. Sono contento di poterli ringraziare in questo contesto.

La mia dattilografa, editrice e critica è stata Salley Oberlin. I suoi suggerimenti per migliorare la qualità e la sua instancabile insistenza sull'ordine e la chiarezza di questo testo mi hanno portato ad impegnarmi per scrivere il miglior libro che io potessi fare.

Roger Gottlieb era stato incaricato di trasformare il manoscritto battuto a macchina, ed una pila di figure schizzate manualmente, in un libro finito. Gli sarò sempre grato per il modo con cui ha cercato di coinvolgermi in tutti gli aspetti della produzione del libro. Un giorno, sono sicuro, sarò anche contento per il modo con cui ha catturato la mia passione per il perfezionismo e la continua ricerca di migliorare se stessi, indirizzandola verso percorsi costruttivi che non hanno interferito né con i soldi né con le tempistiche imposte dal lavoro.

J. Trujillo Smith ha realizzato le illustrazioni che appaiono in questo libro. La sua pazienza ed attenzione nel disegnare e ridisegnare molte di queste illustrazioni hanno permesso di raggiungere un grado di chiarezza notevolissimo. Sono piene di particolari che solo pochi lettori saranno in grado di vederli tutti.

Diana J. Goodin ha realizzato la maggior parte della composizione. Gli algoritmi che ha dovuto escogitare per realizzare alcune mie concezioni, con la sua macchina, sono stati complessi e sofisticati come d'altronde appaiono dal libro.

A Guy S. Orcutt l'ingrato lavoro di integrare le correzioni nella prima

bozza e di registrare le varie versioni che venivano passate dal compositore all'autore per leggerle e controllarle in una sequenza apparentemente senza fine.

Sono anche grato alla Zilog ed AMD per tutto l'aiuto ricevuto durante questo lavoro. David Stevenson e Jim Weldon della Zilog e Steve Dines della AMD sono stati di particolare aiuto nel reperimento di informazioni e per risolvere argomenti dettagliati quando non era sufficiente la documentazione pubblicata disponibile. David Stevenson ha fornito molti suggerimenti utili dopo aver letto il manoscritto originale e Steven Dines mi ha fornito una prima versione della Figura 9.14. Anche la Zilog mi ha fornito la copia del suo lavoro originale sui diagrammi dei piedini dello Z8001 e Z8002. Questi sono riportati in Figura 2.11 per gentile cortesia della Zilog.

Un contributo molto importante al rigore del libro è stato dato da Ken McKenzie, Ron de Jong e Jagi Shahani della Zilog che mi hanno permesso di avere in uso un sistema di sviluppo Z8000, una piastra di valutazione ed un video. Questo mi ha permesso di controllare molti dei programmi riportati in questo libro. Alex Cannara è stato molto gentile mostrandomi come usare questo sistema.

Per la produzione di un libro di questo tipo è molto importante un processo di revisione tecnica. Desidero perciò ringraziare i seguenti professionisti del settore per avere accettato di leggere le copie del manoscritto originale: Dennis Allison, Forrest Baskett, Steve Dines, Costance Hwang, R.S. Langer, Bernard Peuto, David Stevenson, Rodnay Zaks.

Un riconoscimento particolare è dovuto a R.S. Langer, che mi ha suggerito di scrivere un libro sullo Z8000.

Una parola riguardante i vari programmi riportati in questo libro. Il loro scopo è illustrare i principi e le tecniche presentati nel testo. Per chiarezza, non sono stati appesantiti dalle convenzioni speciali di uno specifico assemblatore e sono stati composti e non riprodotti come uscita del calcolatore. Inoltre, per motivi di precisione, molti dei programmi sono stati tradotti in PLZ/ASM — un processo che lascia virtualmente inalterata la parte di codice — e testati. Le routine di commutazione di contesto di Figura 6.21 e i programmi del Capitolo VIII (fino a Figura 8.15) sono stati tutti controllati in questo modo.

Per ultimo lasciatemi dire che le idee ed i concetti presentati in questo libro provengono da varie parti. L'attribuzione a specifiche persone è stata possibile solo in casi particolari, molte di queste idee e concetti sono dovuti a origini sconosciute o multiple; alcune sono semplicemente delle mie interpretazioni personali di principi largamente conosciuti, di scienza del calcolatore e di progetti di sistemi.

Introduzione

Questo libro, da solo, è sufficiente a fornire gli elementi necessari per la programmazione dei microprocessori Z8000:

- Descrive in dettaglio l'architettura ed il funzionamento dello Z8000, mostrando inoltre come questo interagisce con i dispositivi di supporto appartenenti alla sua stessa famiglia.
- Utilizzando lo Z8000, fornisce, come esempio pratico, una introduzione alla programmazione in linguaggio macchina.
- Riporta molti esempi di programmi Z8000 al fine di illustrarne i principi e le tecniche essenziali. Questi programmi sono sviluppati attorno ad argomenti considerati nuclei centrali delle applicazioni come, per esempio, l'interazione operatore-terminale, la gestione della memoria, i programmi condivisi e il time-sharing.
- Fa vedere come possano essere implementati con la programmazione in linguaggio macchina, importanti principi dell'ingegnerizzazione del software come la semplicità, la chiarezza dei commenti, la modularità e così via.

Questo libro è indirizzato a chiunque sia interessato ad usare lo Z8000, ed in particolare si è tenuto conto del fatto che ogni lettore sarà interessato ad aspetti diversi dell'argomento. Per chi desideri imparare il linguaggio macchina di programmazione, il corso base di studio è costituito dalla parte che va dal Capitolo I fino al Capitolo V più una piccola parte del Capitolo VII. I Capitoli II, IV, V e VII comprendono una descrizione dello Z8000 e dei dispositivi di supporto della sua famiglia. Il progettista scarsamente interessato alla programmazione, può leggere questi capitoli per avere una descrizione della macchina ed essere in grado di valutarne i punti di forza e i punti deboli. Chi sarà interessato a continuare, nella parte restante del libro troverà molti esempi, alcuni dei quali potrebbero essere di difficile comprensione per un principiante.

I Capitoli I e III possono essere saltati dai programmatori esperti (in

particolare da quelli che hanno familiarità con CPU analoghe, come per esempio il PDP-11), che in questo caso dovrebbero però studiare l'ultima parte del Capitolo VI ed i Capitoli VIII e IX.

Il libro, prima di descrivere gli esempi di programmazione, riporta alternando opportunamente gli argomenti, un'introduzione alla programmazione e alla descrizione circuitale (hardware).

Il Capitolo I è un'introduzione alla programmazione. In esso si analizzano gli algoritmi, i flussi di programma, ed i vari metodi utilizzati per rappresentare l'informazione all'interno della memoria del calcolatore.

Il Capitolo II consiste in una descrizione generale dell'architettura dello Z8000. Siccome la CPU Z8000 è di tipo avanzato e sofisticato, in questo capitolo sarete obbligati ad analizzare molti concetti difficili. In particolare, per coloro che non hanno mai studiato l'architettura di un calcolatore prima d'ora, una parte di questo materiale apparirà ostica. Se nel cercare di comprendere i dettagli o i significati di questi concetti, incontrate delle difficoltà, limitatevi a scorrere questo capitolo, e rileggetelo poi, dopo aver letto alcuni dei capitoli successivi.

Nel Capitolo III viene illustrato lo sviluppo completo di un programma per lo Z8000 partendo dall'algoritmo iniziale per arrivare fino al codice finale che rappresenta la routine principale e le sue due subroutine. L'algoritmo dato è usato per crittografare un testo mediante l'OR esclusivo eseguito tra i caratteri del testo e dei caratteri chiave. Questo esempio è un interessante sistema con cui presentare concetti, principi e tecniche.

Nel Capitolo IV viene descritto l'insieme delle istruzioni dello Z8000. Per aiutarne la comprensione, le istruzioni sono state raggruppate in otto categorie, che ne facilitano un'esposizione ordinata. È quindi importante capire le istruzioni, e non le categorie. Come ulteriore aiuto viene fornita, solo come strumento di ausilio, una rappresentazione grafica della esecuzione di un'istruzione con il relativo «percorso dei dati».

Il manuale base di riferimento delle istruzioni viene riportato alla fine del Capitolo IV. I 105 codici mnemonici delle istruzioni dello Z8000 sono stati divisi in 45 gruppi; in ogni gruppo sono riportate le istruzioni che essendo fondamentalmente simili in operatività e formato ci è sembrato utile trattare come una unità piuttosto che in modo separato. Alla fine del Capitolo IV ci sono le descrizioni di 45 istruzioni precedute da due indici: uno è relativo alla classificazione alfabetica dei 105 mnemonici di base, ognuno indirizzato alla corrispondente descrizione dell'istruzione; l'altro riguarda una classificazione mnemonica dei 64 «codici operativi» ciascuno indirizzato sia allo mnemonico corrispondente, sia alla descrizione dell'istruzione relativa. Le descrizioni delle illustrazioni ed i loro indici sono riportati, come manuale di riferimento, nel Capitolo IV che comprende an-

che un lungo paragrafo che descrive il loro uso e tratta esempi di assemblaggio e dissemblaggio manuale di programmi dello Z8000.

Il Capitolo V presenta i vari modi di indirizzamento dello Z8000, completando così la descrizione del funzionamento e delle istruzioni della CPU. Questo breve capitolo descrive i principali modi di indirizzamento (registro, indiretto tramite registro, immediato, diretto e indicizzato), i due modi disponibili solamente con l'istruzione di caricamento (base e indicizzato tramite base) ed il modo di indirizzamento relativo implicito. Vengono riportati con un esempio gli argomenti immediati ed il formato dell'indirizzo delle istruzioni. Inoltre viene illustrata la codifica dei campi del registro.

Il Capitolo VI descrive le tecniche di ingresso/uscita. Viene analizzato l'indirizzamento dello I/O e vengono descritte le tecniche per la generazione di livelli ed impulsi, di anelli (loop) di attesa e la trasmissione di bit di dati in modo seriale sincrono. Utilizzando la routine di gestione della telescrivente viene fornito un esempio di trasmissione seriale di bit.

Vengono discusse le trasmissioni half-duplex e full-duplex da terminale e la trasmissione asincrona parallela di dati. Viene analizzato in dettaglio un esempio in cui si descrive l'utilizzo di un visualizzatore a sette segmenti. Infine, viene sviluppata la gestione in interruzione o in polling dello I/O e viene portata come esempio una routine per la gestione dello I/O in interruzione. Quest'esempio include analisi molto dettagliate che possono risultare di difficile comprensione se trattate in modo superficiale.

Il Capitolo VII riguarda i dispositivi periferici dello Z8000, come per esempio l'unità per la gestione della memoria e i dispositivi di I/O seriale e parallelo. L'autore ha utilizzato le migliori informazioni in suo possesso al momento della scrittura del libro, e nelle future edizioni saranno riportati maggiori dettagli. Vengono anche descritte alcune caratteristiche non pianificate dello Z8000 come per esempio l'uso del registro REFRESH, adoperato come base per il clock.

Il Capitolo VIII contiene molti esempi che dimostrano come programmare lo Z8000 e quanto sia facile approntare utili applicazioni. Una di queste, descritta in dettaglio in questo capitolo, è un programma di gestione di un terminale con possibilità di programmazione dell'interazione operatore-terminale (video o telescrivente). Esso utilizza i programmi di interruzione descritti nel Capitolo VI e illustra le routine fondamentali per la gestione delle domande, l'interpretazione delle risposte, il controllo del cursore e la visualizzazione dello schermo. In questo capitolo sono riportati molti altri utili programmi come per esempio la gestione del buffer di linea e d'anello, la traduzione tramite ricerca associativa e la scansione della tabella di corrispondenza.

Il Capitolo IX illustra la potenza dell'architettura dello Z8000 mostrando una semplice implementazione di un sistema in time-sharing che utilizza dei programmi condivisi con stack (pila/catasta) separati per ogni utilizzatore. Vengono anche descritti: una tecnica di gestione dello stack, i dettagli del trattamento della trappola e l'inizializzazione del sistema.

Per ultimo, nel Capitolo X viene descritto l'ambiente per lo sviluppo dei programmi; vengono analizzati dei problemi e fatte delle considerazioni di progetto per editori, assemblatori, caricatori e per le prestazioni del debugger.

Vengono analizzate, anche, le possibili alternative circuitali valide per lo sviluppo del software; per esempio sistemi di sviluppo e «calcolatori su singola scheda».

Questo è un lungo libro contenente molti esempi concreti di programmi ed utilizzo delle istruzioni. Nonostante l'autore abbia cercato in ogni modo di fornire un testo corretto, si potranno trovare inevitabilmente degli errori. Sarà notevolmente apprezzata ogni segnalazione di tali errori, commenti e critiche che, in successive edizioni contribuiranno sicuramente al miglioramento del libro.

Capitolo I

Concetti Base

In questo capitolo verranno descritti gli algoritmi, i diagrammi di flusso ed il modo con cui vengono rappresentate le informazioni. Il lettore che conosce già questi argomenti può limitarsi a scorrerli molto velocemente.

Che cosa è la programmazione?

La programmazione è un'arte che, come la carpenteria o la scrittura, richiede oltre alla conoscenza degli elementi di base, anche una discreta esperienza pratica. Noi abbiamo molto materiale grezzo su cui lavorare e alcuni lavori da eseguire. Il materiale grezzo è costituito dalle istruzioni, dalla memoria e dai dispositivi periferici del nostro sistema calcolatore, ossia lo Z8000. Nel momento in cui andremo ulteriormente avanti, dovremo avere una idea migliore di come sono fatti i lavori, ma per ora, diciamo che siamo interessati a risolvere i problemi.

Algoritmi

La prima cosa che ognuno impara sui calcolatori è che essi fanno ciò che gli si chiede di fare, e voi, onde evitare di sbagliare, dovrete esprimervi, verso questi, in modo chiaro e semplice, altrimenti farete la fine dell'uomo che chiese al genio di fargli un frullato. Iniziamo con un algoritmo che stabilisce esattamente che cosa deve fare il calcolatore: ossia fissiamo un elenco di istruzioni, chiamate passi, numerate secondo l'ordine in cui vengono eseguite. Per far sì che l'algoritmo sia utile abbiamo bisogno di definire dei «passi» del tipo: «Se è così e così, allora salta avanti (o indietro) ed esegui così e così». Questo complica le cose. Infatti, per evitare di ripetere ciclicamente lo stesso insieme di istruzioni dobbiamo prevedere una via d'uscita dal ciclo.

Nella Figura 1.1 viene illustrato il modo abituale in cui è eseguita la

12573	———	Passo 1
112		
25146	———	Passo 4, prima volta (n = 1)
12573	———	Passo 4, seconda volta (n = 2)
12573	———	Passo 4, terza volta (n = 3)
1408176	———	Passo 7

Figura 1.1 — Moltiplicazione di Due Numeri

moltiplicazione di due numeri. Per darvi un'idea di come si parli ad un calcolatore, nella Figura 1.2 è riportato l'algoritmo corrispondente. Ogni volta che abbiamo un nuovo problema da risolvere, dobbiamo sviluppare un algoritmo e farlo conoscere al calcolatore.

Vediamo come funziona in realtà l'algoritmo di Figura 1.2 che si riferisce all'esempio di Figura 1.1. Iniziamo scrivendo, al primo passo, 112 sotto 12573, in corrispondenza delle ultime tre cifre a destra, e poi tracciamo una linea sotto di esse.

Il secondo passo è definito «*inizializzazione del contatore*». Nella figura non appare da nessuna parte, ma è solo una convenzione per ricordarci con quale delle tre cifre di 112 stiamo lavorando. Ci sposteremo da destra verso sinistra, per cui n = 1 indica che stiamo lavorando con 2; n = 2 indica la cifra di mezzo 1; e n = 3 indica la prima cifra di 112, cioè 1.

1. Scrivete i due numeri uno sotto l'altro con la loro ultima cifra incolonnata e tracciate una riga sotto di essi.
2. Ponete il contatore a 1.
3. Se il valore n del contatore, supera il numero di cifre contenute nel moltiplicatore, saltate al passo 7.
4. Moltiplicate il primo numero per la n-esima cifra, partendo dall'estremo destro del moltiplicando, e scrivete la risposta sotto la linea tracciata, con la cifra più a destra incolonnata con la cifra n-esima del moltiplicando.
5. Incrementate di 1 il contatore (esempio sostituire n con n + 1).
6. Tornate indietro al passo 3.
7. Sommate tutti i numeri ottenuti ripetendo il passo 4; questo è il risultato richiesto.

Figura 1.2 — Algoritmo della Moltiplicazione

Il terzo passo è definito «*controllo di una condizione*». Confronteremo n, che abbiamo appena posto uguale a 1, con 3, il numero di cifre di cui è composto 112. Naturalmente, 1 non è maggiore di 3, e allora non salteremo al passo 7. Quando eseguiamo i passi 5 e 6, portiamo il valore di n a 2, poi a 3 e poi a 4 ed ogni volta torniamo indietro a confrontarlo con il valore 3. Quando finalmente n diviene uguale a 4, ossia supera il valore 3, saltiamo al passo 7. I passi 2, 3, 5 e 6 presi insieme sono equivalenti a: esegui una volta il passo 4 per ogni cifra del moltiplicando, perciò nel caso di 112,3 volte. I passi da 2 a 6 costituiscono il cosiddetto *anello (loop)*. Il passo 4 definisce ciò che si vuole fare nell'anello e i passi 3, 5 e 6 sono degli ausigli meccanici che noi usiamo per fare sì che il passo 4 venga eseguito un numero corretto di volte.

Il passo 4, il cuore del loop, ci dice di moltiplicare 12573 per una qualsivoglia cifra di 112 che sia stata raggiunta nel processo di moltiplicazione, e di scriverne la risposta con l'ultima cifra incolonnata con la cifra del moltiplicando appena usata. Naturalmente, la ragione per cui viene eseguito questo allineamento è che i due «1» in 112 rappresentano 100 e 10; quindi l'incolonnamento delle risposte in corrispondenza di queste cifre ci evita di scrivere gli zeri finali. In realtà, la somma sotto la linea di Figura 1.1 potrebbe essere scritta come:

$$\begin{array}{r} 25146 \\ 125730 \\ 1257300 \\ \hline 1408176 \end{array}$$

Se volete che qualcuno moltiplichi 12573 per 112, non fate una serie di sette domande, ma chiedete solo: «Quanto fa 12573 per 112?». Se la persona ha imparato (e ricorda) l'algoritmo, questa domanda concisa farà avere il risultato desiderato.

Analogamente, una volta che è stato comunicato un algoritmo al calcolatore è sufficiente inviare una semplice domanda al calcolatore per chiedergli di applicare l'algoritmo al caso in esame. Questo è ciò che fanno le subroutine; perciò l'eliminazione di ripetizioni è una delle ragioni per cui le subroutine sono l'unico strumento importante del programmatore.

ESERCIZIO 1: Tutti i passi dell'algoritmo della moltiplicazione di Figura 1.2 non vengono ripetuti lo stesso numero di volte. Quando si applica l'algoritmo al caso 12573 per 112, quante volte viene eseguito il passo 3? Uno dei passi viene eseguito una sola volta? Che cosa succede se scriviamo 112 sopra e 12573 sotto?

L'algoritmo parte dall'estremo destro del numero riportato sotto, e si

sposta verso sinistra. Siete in grado di realizzare un algoritmo che parte dall'estremo sinistro e si sposta verso destra? In quale modo realizzerete il giusto incolonnamento dei numeri? Provate il vostro algoritmo con 9536 × 121. Supponendo di essere interessati ad avere solamente una risposta approssimata; esiste un vantaggio reale a partire da sinistra?

Diagrammi di flusso

Il diagramma di flusso è un altro modo di rappresentare un algoritmo. In parole povere il diagramma di flusso è la rappresentazione grafica di un algoritmo. I vari passi sono rappresentati da rettangoli mentre le linee e le frecce indicano la sequenza dei passi. I passi che significano: «se è così e così salta (o torna indietro) al passo tal dei tali» sono disegnati come rombi, e sono caratterizzati dall'avere più di una linea in uscita. In Figura 1.3 è riportato il diagramma di flusso dell'algoritmo della moltiplicazione. I rettangoli e le linee del diagramma sono indicati con i numeri corrispondenti ai passi dell'algoritmo.

Per concludere, si può dire che non importa che, per esprimere un algoritmo, voi utilizziate il diagramma di flusso o la forma descrittiva, è però importante, *prima* di scrivere un programma usare uno di questi metodi.

ESERCIZIO 2: Disegnate il diagramma di flusso per l'apertura della vostra porta principale supponendo che in un senso sia chiusa a chiave e nell'altro no.

Rappresentazione dell'informazione

Un calcolatore è rappresentabile mediante tre blocchi fondamentali: l'unità centrale di elaborazione (CPU) che esegue tutto il lavoro; il programma memorizzato che non è altro che l'algoritmo espresso in termini di operazioni e punti decisionali che la CPU è capace di eseguire; e la memoria per memorizzare i dati su cui la CPU eseguirà il proprio lavoro.

Rappresentazione esadecimale

Prima di discutere come funzioni la CPU ed i programmi memorizzati, analizzeremo in che modo è possibile rappresentare i vari tipi di dati nella memoria. La memoria di tutti i calcolatori è costituita da bit, o cifre binarie, che sono elementi a due stati rappresentati sempre tramite 0 o 1. Immaginateli come piccole lampadine: lo zero è equivalente a spento e uno ad acceso. Nello Z8000 il raggruppamento principale di bit è la *parola* di 16-bit; le parole, a loro volta sono divisibili in due *byte* di 8-bit. Ciascun byte è costituito da due *cifre* di 4-bit. Muovendosi nell'altra direzione, si trova che, qualche volta, due parole vengano unite per costituire una *pa-*

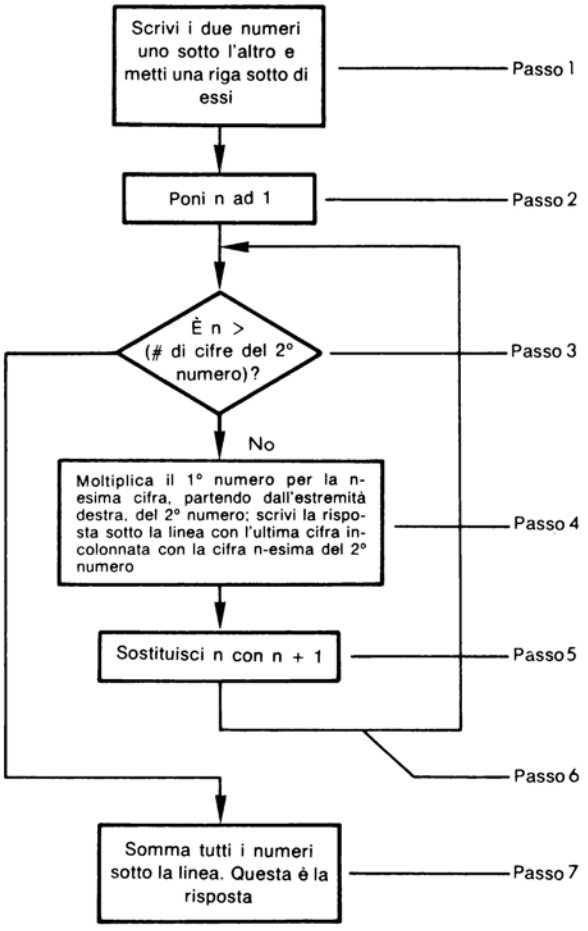


Figura 1.3 — Diagramma di Flusso dell'Algoritmo della Moltiplicazione

rola lunga di 32-bit. Nella CPU Z8000 vengono occasionalmente riconosciute anche entità di 64-bit. Lo stato reale dei bit (le luci) di ciascuno dei raggruppamenti è denominato *bit pattern* (configurazione della cifra binaria) *o valore* del byte, della parola, etc. (vedasi Figura 1.4).

Noi rappresentiamo i dati in memoria facendo corrispondere significati diversi a diverse configurazioni di bit. Qualche volta la medesima configurazione di bit, assume nel tempo significati diversi in funzione del fatto che può essere interpretata come un numero, un carattere di un testo o qualche cosa altro. In effetti, una delle cause di errore nei programmi scritti in linguaggio macchina, ma anche in molti programmi scritti con

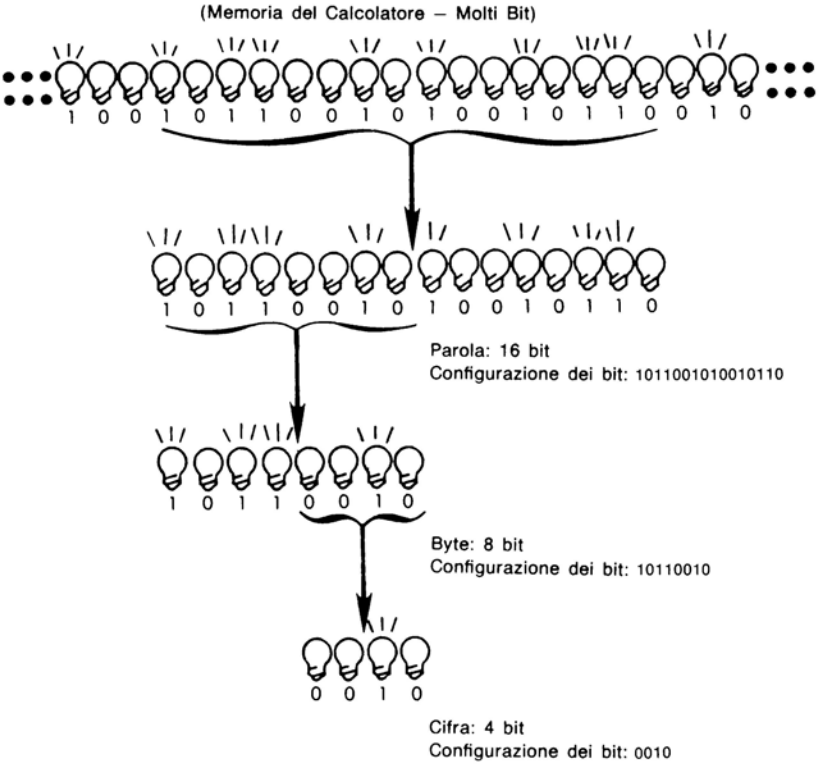


Figura 1.4 — Configurazioni e Raggruppamento di Bit

linguaggi «evoluti», sta nell'interpretare erroneamente un certo tipo di dato in memoria.

Dal momento che ogni cosa dipende da come si interpreta la configurazione dei bit, dobbiamo trovare un modo per rappresentarli. Il modo più comune è la rappresentazione *esadecimale* in base 16. Nella Figura 1.5 si riportano i valori corrispondenti a ciascuna delle combinazioni possibili ottenibili con quattro bit. Si può constatare come queste combinazioni siano ordinate secondo una certa logica e che i nomi non sono assegnati in modo casuale. Le configurazioni di bit sono ordinate ed interpretate come *numeri binari.*

Le rappresentazioni binaria ed esadecimale sono due esempi di *sistemi di numerazione.* Nella nostra rappresentazione abituale, col sistema numerico *decimale*, un qualsiasi numero può essere scritto esattamente solo in un modo:

$$a_0 + a_1 \times 10 + a_2 \times 100 + a_3 \times 1000 + ...$$

Configurazione	Nome	Configurazione	Nome
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Figura 1.5 — Nomi delle Configurazioni Esadecimali dei Bit

equivalente a

$$a_0 + a_1 \times 10 + a_2 \times 10^2 + a_3 \times 10^3 + ...$$

dove a_0, a_1, a_2, sono numeri interi compresi tra 0 e 9. Per esempio, 1980 è la forma abbreviata di

$$0 + 8 \times 10 + 9 \times 100 + 1 \times 1000$$

Analogamente, se b è un qualsiasi numero intero maggiore di 1, si può dimostrare che può essere scritto esattamente in un solo modo come:

$$c_0 + c_1 \times b + c_2 \times b^2 + c_3 \times b^3 + ...$$

dove c_0, c_1, ...sono numeri interi compresi tra 0 e b-1. Per esempio:

$$196 = 4 + 0 \times 8 + 3 \times 8^2$$

Per b = 2 e b = 16, i sistemi di numerazione corrispondenti sono denominati binario ed esadecimale. Quando b = 2, i coefficienti c_0, c_1... prendono tutti valore 0 o 1; in tal modo una qualsiasi configurazione di bit può essere vista come un numero binario.

Quando b = 16, i coefficienti c_0, c_1, ...assumono dei valori compresi tra 0 e 15. Per rappresentarli, useremo per convenienza le seguenti abbreviazioni:

$$A = 10;\ B = 11;\ C = 12;\ D = 13;\ E = 14;\ F = 15.$$

Per esempio:

$$196 = 4 + C \times 16$$

Scriviamo la rappresentazione in base b, per analogia con la nostra rappresentazione decimale:

$$c_0 + c_1 \times b + c_2 \times b^2 + c_3 \times b^3 + \ldots$$

come

$$\ldots c_3 c_2 c_1 c_0$$

Per esempio,

$$196 = C4$$

nella rappresentazione esadecimale.

Dal momento che alcuni numeri esadecimali sono rappresentati come numeri decimali, quando si può avere confusione nell'interpretazione scriveremo il numero base come indice. Per esempio,

$$196_{10} = 304_8 = C4_{16}$$

ESERCIZIO 3: Si scriva 7 in codice binario. Si traduca 196 in codice binario. Quale è la relazione tra la rappresentazione di 196 in codice binario ed esadecimale?

A questo punto diventano evidenti i nomi assegnati in Figura 1.5 alle diverse configurazioni di bit: il nome assegnato a ciascuna configurazione di 4-bit è la cifra decimale corrispondente al numero binario di 4 bit che esso rappresenta. Per esempio,

$$1101 = 1 + 0 \times 2 + 1 \times 2^2 + 1 \times 2^3 = 13 = D_{16}$$

Inoltre, spezzando le configurazioni in gruppi di 4 bit (partendo da destra se il numero di bit della configurazione data non è multiplo di 4) e riportando per ciascun gruppo il suo nome esadecimale è possibile estendere questa rappresentazione esadecimale a configurazioni di 8 bit, 16 bit, o più lunghe. Se viene fatto quanto detto sopra, il *nome* esadecimale risultante, considerato come *numero* esadecimale, rappresenta lo stesso numero che corrisponde al numero binario della configurazione originale di bit. Nella Figura 1.6 è riportato un esempio. Si noti come il numero binario, sul lato destro del diagramma sia valutato raggruppando insieme i termini per 16, 32, 64 e 128 e raccogliendo poi il fattore 16. Ovviamente, per provare quello che si è detto sopra, è possibile estendere questo metodo a configurazioni di bit più lunghe di quelle riportate in Figura 1.6.

ESERCIZIO 4: Si usi il metodo di Figura 1.6 per mostrare che $11000100_2 = 304_8$.

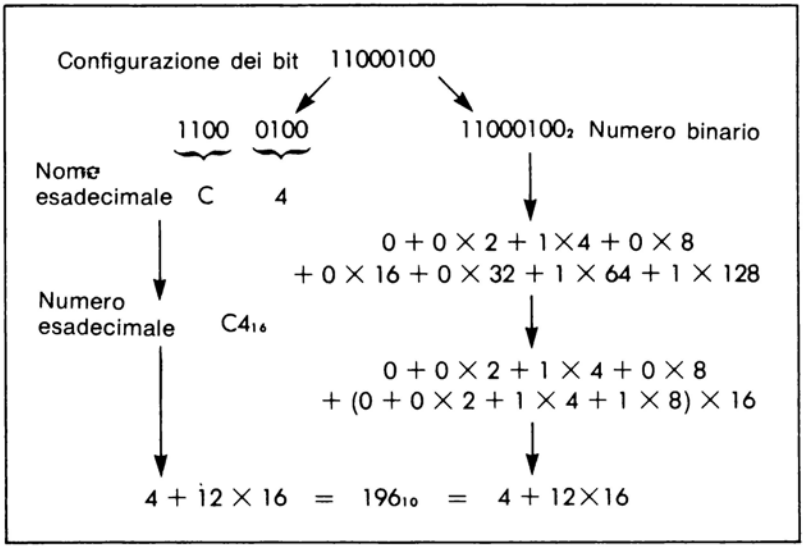


Figura 1.6 — Relazione tra Rappresentazione Binaria ed Esadecimale

Se consideriamo il numero base b limitato ad n cifre, possiamo esprimere un qualsiasi numero compreso tra 0 e $b^n - 1$ nella forma:

$$c_0 + c_1 \times b + c_2 \times b^2 + ... + c_{n-1} \times b^{n-1}$$

Per esempio, si può esprimere un qualsiasi numero compreso tra 0 e 999 (essendo $999 = 10^3 - 1$) limitandosi a numeri decimali di 3 cifre. Dal momento che per distinguere tra di loro gli elementi di un gruppo occorre assegnare a ciascuno di essi un numero, e siccome ci sono b^n numeri tra 0 e $b^n - 1$, esiste un altro modo per esprimere tutto questo e cioè: n cifre in base b possono rappresentare b^n elementi distinti. In particolare si ha la seguente importante regola:

Regola: (a) n bit possono rappresentare 2^n elementi differenti.
(b) n cifre esadecimali possono rappresentare 16^n elementi differenti.

Per esempio, per due bit si ha $2^2 = 4$ configurazioni di bit: 00, 01, 10, 11. Possiamo distinguere quattro elementi differenti facendo corrispondere ad ognuno di essi una delle configurazioni. (Vedasi Figura 1.7).

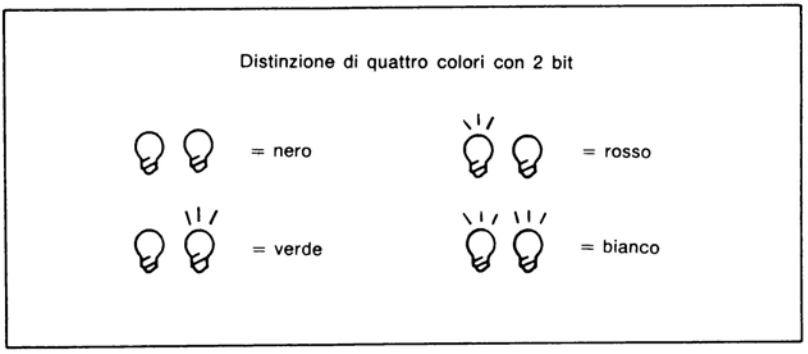


Figura 1.7 — Uso di n Bit per Distinguere 2^n Elementi

ESERCIZIO 5: Quanti elementi diversi si possono distinguere con 3-bit? Quanti bit occorrono per distinguere 27 elementi diversi? Se state progettando un microprocessore con 12 stati diversi, che deve comunicare con il mondo esterno, quante linee di stato (bit) dovete usare per poter assegnare una diversa configurazione di bit a ciascuno di essi? Quanti numeri diversi si possono rappresentare con 4 cifre esadecimali? Con 8?

Dal momento che all'interno del calcolatore i numeri devono essere rappresentati in forma esadecimale mentre nel mondo reale sono generalmente rappresentati in forma decimale, dovremo imparare come si fà a passare da una forma all'altra. Siccome sappiamo come eseguire operazioni aritmetiche in forma decimale, si può facilmente passare dalla forma esadecimale a quella decimale utilizzando la definizione di rappresentazione esadecimale. Per esempio:

$$C4_{16} = 12 \times 16 + 4 = 192 + 4 = 196$$

Nella Figura 1.8 si riporta un algoritmo per passare dalla rappresentazione decimale a quella esadecimale utilizzando solo l'aritmetica decimale. Pur sembrando un algoritmo complesso è in realtà molto semplice. Nella Figura 1.9 viene riportato un esempio di applicazione di questo algoritmo al numero decimale 299. Ad ogni livello viene eseguita una divisione per 16. Il resto costituisce la prossima cifra esadecimale (muovendosi da destra a sinistra) e il quoziente costituisce il punto di partenza del livello successivo. L'ultima equazione mostra come funziona; il numero esadecimale $h_3h_2h_1h_0$ può essere scritto come:

$$((((0 \times 16 + h_3) \times 16 + h_2) \times 16 + h_1) \times 16 + h_0$$

Assumi che sia stato assegnato un numero decimale $d_n d_{n-1} \ldots d_0$. Vogliamo trovare i coefficienti $h_m, \ldots, h_0$ in modo che

$$(h_m \ldots h_0)_{16} = (d_n d_{n-1} \ldots d_0)_{10}$$

1. Poni il valore del contatore **k** eguale a 0. Poni $N = d_n d_{n-1} \ldots d_0$.
2. Dividi N per 16_{10}. Assumi che h_k sia il resto ed il nuovo valore di N sia il quoziente.
3. Se N = 0 fermati; $(h_k \ldots h_0)_{16}$ è la rappresentazione voluta. Altrimenti incrementa di **1** il valore di k e salta al Passo 2.

Figura 1.8 — Algoritmo per la Conversione da Decimale a Esadecimale

Dividendo per 16 si ha come resto h_0 e come quoziente

$$((0 \times 16 + h_3) \times 16 + h_2) \times 16 + h_1$$

Dividendo questo per 16 il resto è h_1 e il quoziente

$$(0 \times 16 + h_3) \times 16 + h_2$$

La prossima divisione dà h_2 come resto e quoziente

$$0 \times 16 + h_3$$

L'ultima divisione dà come resto h_3. Questo stesso schema può essere esteso ad una rappresentazione esadecimale composta da un generico numero di cifre.

ESERCIZIO 6: Si applichi l'algoritmo di Figura 1.8 al numero decimale 196.

Se si desidera sommare, sottrarre, moltiplicare o dividere due numeri esadecimali, è possibile convertirli in numeri decimali, eseguire l'operazione e poi riconvertire il risultato in esadecimale. Ovviamente l'approccio diretto consiste nel realizzare le operazioni aritmetiche direttamente in esadecimale. Nella Figura 1.10 sono riportate le tabelle di addizione e moltiplicazione per numeri esadecimali composti da una sola cifra. L'estensione dell'addizione e della moltiplicazione a numeri di più cifre avviene e-

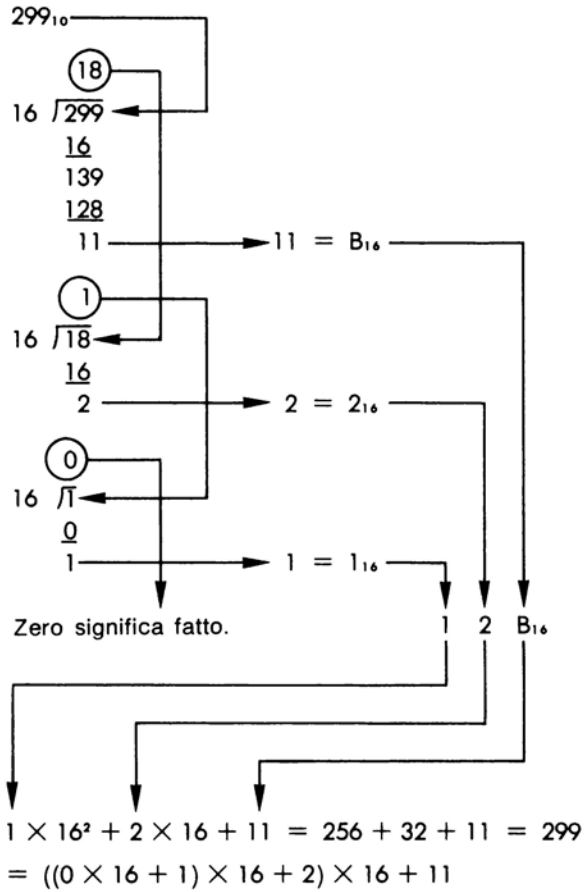


Figura 1.9 — Conversione di 299_{10} in $12B_{16}$

sattamente come per i numeri decimali: se la somma o il prodotto di due numeri di una cifra è un numero di due cifre, la cifra di valore più elevato viene «spostata» a sinistra di una posizione. La sottrazione e la divisione sono ricavate, esattamente come per i numeri decimali, dall'addizione e dalla moltiplicazione.

Supponiamo, per esempio, di voler moltiplicare $7A_{16}$ per 26_{16}. Nella Figura 1.11 vengono riportate questa moltiplicazione e la corrispondente moltiplicazione decimale. Si inizia moltiplicando 7A per 6. Per prima cosa si moltiplica A per 6 e per far questo si ricerca nella tabella della moltiplicazione di Figura 1.10 l'intersezione della riga A con la colonna 6, si trova 3C. Si scrive C e si riporta 3. Poi si trova che $7 \times 6 = 2A$. Sommia-

ADDIZIONE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0
		2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	1
			4	5	6	7	8	9	A	B	C	D	E	F	10	11	2
0	0			6	7	8	9	A	B	C	D	E	F	10	11	12	3
1	0	1			8	9	A	B	C	D	E	F	10	11	12	13	4
2	0	2	4			A	B	C	D	E	F	10	11	12	13	14	5
3	0	3	6	9			C	D	E	F	10	11	12	13	14	15	6
4	0	4	8	C	10			E	F	10	11	12	13	14	15	16	7
5	0	5	A	F	14	19			10	11	12	13	14	15	16	17	8
6	0	6	C	12	18	1E	24			12	13	14	15	16	17	18	9
7	0	7	E	15	1C	23	2A	31			14	15	16	17	18	19	A
8	0	8	10	18	20	28	30	38	40			16	17	18	19	1A	B
9	0	9	12	1B	24	2D	36	3F	48	51			18	19	1A	1B	C
A	0	A	14	1E	28	32	3C	46	50	5A	64			1A	1B	1C	D
B	0	B	16	21	2C	37	42	4D	58	63	6E	79			1C	1D	E
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90			1E	F
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9			
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4		
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

MOLTIPLICAZIONE

Figura 1.10 — Tabella di Addizione e Moltiplicazione Esadecimale

mo a 2A il riporto 3 e si ottiene 2D. Scriviamo questo a fianco della prima cifra C ottenendo 2DC. Analogamente, 7A × 2 = F4; si scrive questo sotto 2DC con il 4 incolonnato con il 2 di 26. Per ultimo si sommano i numeri sotto la linea. F4 in verità è F40; C + 0 = C; D + 4 = 1 con riporto di 1; 2 + F + riporto di 1 = 12. La risposta perciò è: 121C.

ESERCIZIO 7: Quanto fa B + B? E 1A + B1? A^2? Si valuti 2B5 $\sqrt{1B93F}$.

Complemento a due

Finora si è parlato di come, per esempio, una parola di 16 bit possa essere utilizzata per rappresentare i numeri tra 0 e 65535 (essendo $2^{16} = 65536$). Ma, naturalmente, si potrebbe essere interessati a rappresentare anche numeri negativi. Questo viene realizzato tramite un elegante trucco.

Il modo ovvio per gestire i numeri negativi è di usare uno dei bit per rappresentarne il segno. Per esempio, si potrebbe usare il bit più a sinistra, di una parola a 16-bit, come bit di segno ed i rimanenti 15 bit per rappresentare grandezze comprese tra 0 e 32767. Questo metodo è conosciuto come rappresentazione *dell'ampiezza con segno.* Non viene utilizzato nello Z8000 (né in altri calcolatori) perché è più difficile l'implementazione circuitale rispetto al metodo realmente utilizzato, inoltre presenta l'ulteriore svantaggio di avere due rappresentazioni dello zero: -0 e $+0$.

Esadecimale	**Decimale**
7A	119
26	34
2DC	476
F4	357
121C	4046

Figura 1.11 — Moltiplicazione Esadecimale

Il metodo che di fatto si usa per rappresentare numeri negativi è denominato *complemento a due.* Il complemento a due «preleva» un bit extra per poi scartarlo, per cui nella rappresentazione in complemento a due, di parole di 16-bit, $-x$ è rappresentato da $2^{16} - x$; i numeri maggiori di $2^{15} - 1$ vengono considerati negativi. Il vantaggio più evidente di questa rappresentazione è che i numeri in complemento a due possono essere sommati come se fossero tutti positivi ottenendo la risposta corretta, siccome:

$$x + (2^{16} - x) = 2^{16} = 10000_{16}$$

Se si considerano solo gli ultimi 16 bit di 10000_{16}, si ha 0000_{16}, per cui

$$x + (-x) = 0$$

come richiesto. Questo è esattamente analogo all'odometro a 5 cifre dell'automobile: 99999 miglia + 1 miglio = 0 miglia. Perciò in un odometro

$$-1 = 100000 - 1 = 99999$$

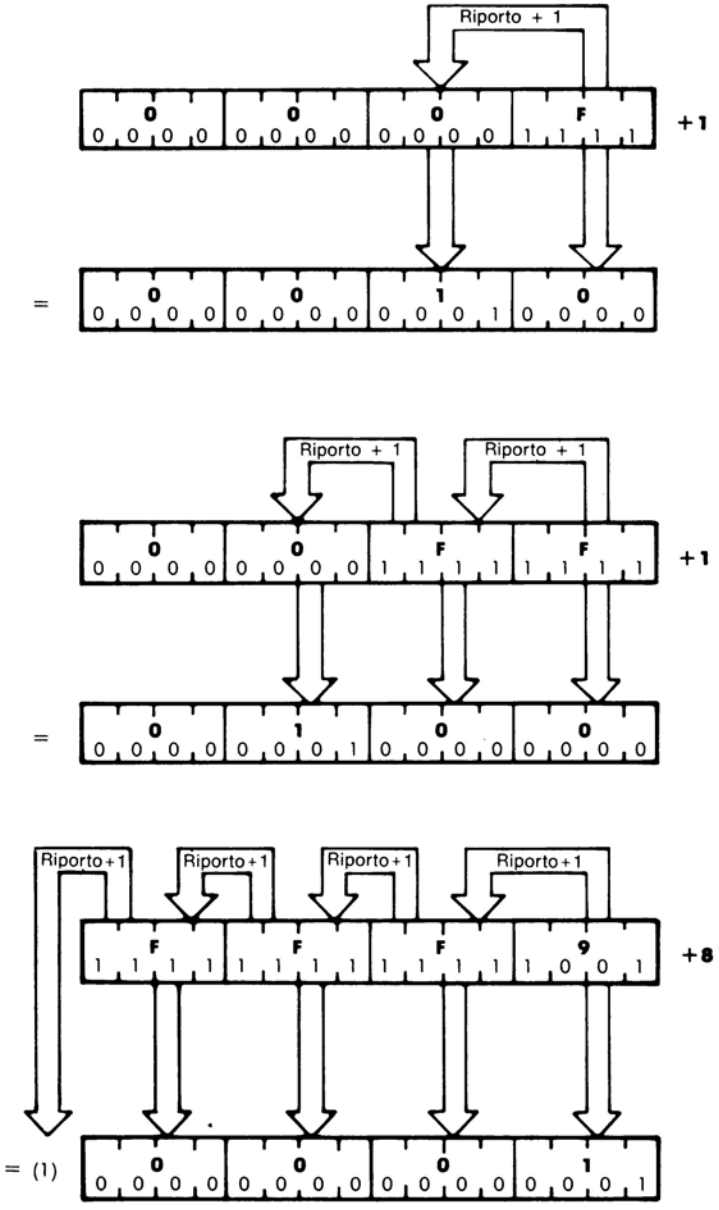


Figura 1.12 — Una Parola di 16-Bit Interpretata come Odometro Esadecimale

Una parola di 16 bit è come un odometro binario (o odometro esadecimale). (Vedasi Figura 1.12).

Vengono riportati in seguito, alcuni esempi di rappresentazione di 16 bit in complemento a due usate nello Z8000: $-1 = \text{FFFF}$; $-20 = \text{FFE0}$; $-8000 = 8000$.

ESERCIZIO 8: Come viene rappresentato -5 in complemento a due? Come è rappresentato -7FFF? Si sottragga 1A1 da FF7 calcolando il complemento a due di -1A1 e sommandolo poi a FF7.

Che cosa si ottiene sommando -1 a -5? Che cosa si ha se si somma -7FFE a -6?

Nel Capitolo II analizzeremo i registri e i bit di condizione. Quando lo Z8000 esegue delle sottrazioni o addizioni predispone dei bit di condizione in modo da far conoscere al programmatore cosa succede. Uno di questi, il bit C, è il diciassettesimo bit che noi prendiamo in prestito per far funzionare il complemento a due. Ogni volta che si esegue una somma di due numeri di 16-bit, il diciassettesimo bit è definito *carry* (riporto). Lo Z8000 ci dà gli ultimi 16 bit come risultato della nostra addizione, ma salva nel bit C il valore del diciassettesimo bit, nell'eventualità che noi volessimo conoscerlo.

Nel caso della sottrazione, il bit C, è gestito nel modo opposto. C viene posto ad 1 (settato) se c'è stato «prestito» e azzerato se non si è verificato nulla; per esempio, per $(4 - 5)$ viene settato C, e per $(5 - 4)$ C non viene settato. La ragione per cui si dice che questo caso è l'opposto di quello dell'addizione è che $4 - 5$ e $4 + (-5)$ danno valori diversi di C.

ESERCIZIO 9: Per ciascuno dei seguenti calcoli si dia la risposta ed il valore del bit C. $6 - 4$; $\text{BB} - \text{BC}$; $\text{FFFF} + 3$; $\text{FFFF} + \text{FFFF}$.

Gli esercizi hanno evidenziato il problema dell'aritmetica in complemento a due: il *superamento della massima capacità aritmetica* rappresentabile (overflow). Per esempio,

$$7\text{FFE} + 6 = 8004 = -7\text{FFC}$$

In altre parole (in decimale)

$$32766 + 6 = -32764$$

In generale si ha un superamento della massima capacità aritmetica rappresentabile quando la somma di due numeri positivi è negativa o quando la somma di due numeri negativi è positiva. Quando si verifica questo, nello Z8000 viene settato il bit V; altrimenti viene azzerato. Que-

sto può essere visto nell'esempio del nostro odometro a 5-cifre. Tutte le letture comprese tra 50000 e 99999 rappresentano numeri negativi, perciò quando si somma il numero positivo 5 al numero positivo 49998 si ha 50003 che è in realtà il numero negativo —49997. Questo è un superamento di capacità (overflow). Con la parola di 16-bit, la fascia 8000 — FFFF corrisponde alla fascia 50000 - 99999 dell'odometro. Non c'è niente di strano a ottenere 50003 sommando 5 a 49998 oppure ottenere 8003 sommando 7FFE a 5. Il problema nasce quando si debbono *interpretare* come numeri negativi 50003 e 8003. In realtà ci sono dei casi in cui vogliamo dimenticarci il complemento a due e considerare il campo 0000 - FFFF come un campo di numeri positivi corrispondente al campo decimale da 0 - 65535. In particolare, questo è vero, quando si interpreta un numero di 16-bit come indirizzo di memoria o come spiazzamento (offset), argomenti che discuteremo in capitoli successivi. Questo è un altro dei motivi per eliminare l'uso della rappresentazione dell'ampiezza di un numero con il segno.

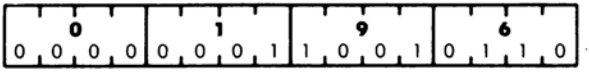
ESERCIZIO 10: Indicare il risultato e il valore dei bit C e V per: 6 — 4; BB — BC; FFFF + FFFF; —17 + (—7FF1).

Rappresentazione BCD

Nello Z8000 viene supportata la rappresentazione in complemento a due per i numeri di 8-bit, 32-bit come anche per quelli a 16-bit. Per i numeri di 8 bit (byte) si ha un campo compreso tra —128 e +127; per i numeri a 16-bit il campo è da —32768 a +32767; infine per i 32-bit il campo è —2147483648 e +2147483647. Questi campi di numeri sono più che soddisfacenti per la maggior parte dei nostri fabbisogni aritmetici, ma qualche volta è più conveniente la *rappresentazione BCD*. BCD significa *Binary Coded Decimal* (decimale codificato in binario), ma è in realtà un decimale codificato in codice esadecimale.

Usando la rappresentazione BCD il calcolatore memorizza le singole cifre di un numero decimale in cifre esadecimali consecutive. Per esempio, il numero decimale 196 viene memorizzato nel calcolatore come 1,9 e 6 in cifre esadecimali consecutive; perciò si scrive 196_{16} (che equivarrebbe a 406_{10}) ma che significa 196_{10} (che è in realtà $C4_{16}$). (Si veda Figura 1.13).

Perché si fa questo? Perché questo è il modo con cui i numeri generalmente entrano ed escono dal calcolatore. Se nel calcolatore entrano 1,9 e 6 è più facile memorizzarli così come sono piuttosto che convertirli in C4 (come vorrebbe la rappresentazione in complemento a due). Se per esempio si vuole visualizzare il numero 196 su uno schermo CRT si deve spez-



Rappresentazione BCD di 196_{10}

Rappresentazione in complemento a due di 196_{10}

Figura 1.13 — Rappresentazione BCD

zarlo in 1,9 e 6; per cui è più facile farlo se lo si rappresenta in questa forma.

ESERCIZIO 11: Disegnate il diagramma di flusso (o scrivete l'algoritmo) che risolva il seguente problema: Si ricevono delle cifre decimali ad una ad una. La sequenza finisce quando si riceve un «X» invece di un numero. Si deve memorizzare un numero esadecimale N in modo tale che quando si riceve il carattere «X», N contenga il valore esadecimale del numero decimale rappresentato dalla stringa di cifre precedenti il carattere «X». Per esempio, se la sequenza dei valori che si riceve è 1, 9, 6, X, partendo da zero N assume i seguenti valori esadecimali 1, 13, C4 (corrispondenti decimali di 1, 19, 196).

Dopo tutte le complicazioni che si creano passando dalla rappresentazione esadecimale a quella decimale e viceversa, vi potreste stupire del perché alla fine si usi la rappresentazione esadecimale. Perché non rappresentare sempre tutto in BCD? La risposta è che lo fanno alcuni calcolatori (esempio, IBM 1401), ma la rappresentazione binaria o esadecimale è più facile da implementare circuitalmente e rappresenta un modo molto più efficiente di utilizzo della memoria. Nella rappresentazione BCD si usano 4 bit per rappresentare le 10 cifre decimali. Le configurazioni di bit corrispondenti ad A, B, C, D, E ed F sono inutilizzate; perciò si perde il 37,5% delle possibili configurazioni di bit.

ESERCIZIO 12: Quanti numeri BCD si possono rappresentare con 16 bit? Quanti numeri esadecimali? Qual è il rapporto di queste due quantità? In quale relazione sta con il rapporto 10/16 valido per quattro bit? Quale è la regola generale?

Nella rappresentazione BCD esistono altri due problemi: uno consiste nel fatto che non c'è una rappresentazione ovvia dei numeri negativi; l'altro che lo Z8000 non ha istruzioni per eseguire operazioni aritmetiche

con numeri in BCD (ad eccezione della *correzione decimale* — vedasi Capitolo IV). Si provi a sommare i numeri 25 e 19 rappresentati in BCD per valutare perché occorrono delle istruzioni speciali. La risposta è 3E che non è la rappresentazione in BCD di 44.

Il comando di correzione decimale è fornito proprio per questi casi. Può essere applicato al risultato sommando numeri BCD di 2-cifre, onde ottenere il risultato corretto. Per esempio, dopo aver sommato 19 e 25, la correzione decimale convertirà 3E in 44.

Oltre al bit di riporto (carry) C, esiste il bit H di semi-riporto (half-carry) che viene posto ad uno dallo Z8000 quando c'è un riporto passando dal digit (cifra) meno significativo di un byte a quello più significativo.

La correzione decimale somma un «fattore correttivo» al byte, poi se c'è stato un riporto decimale, pone ad uno C, per esempio se la risposta supera 99_{10}.

ESERCIZIO 13: Si scriva un algoritmo per la correzione decimale. Esiste un altro bit, D, che lo Z8000 pone ad uno dopo aver *sottratto* dei byte, ma che azzera dopo aver sommato dei byte. Modificate il vostro algoritmo, utilizzando D, in modo che venga eseguita la correzione sia per le somme che per le sottrazioni.

Ora abbiamo imparato cosa sono i bit di stato C, V, D e H; rimangono ancora Z, P e S. Lo Z8000 pone questi bit a uno o a zero in corrispondenza al verificarsi delle varie condizioni.

I simboli a forma di rombo, presenti nei diagrammi di flusso, sono utilizzati per controllare se lo stato di questi bit è uno o zero.

Rappresentazione in virgola mobile

Per ultimo discuteremo la rappresentazione di numeri in *virgola mobile* (floating point). Questa è analoga alla «notazione scientifica» usata sui calcolatori quando la risposta necessita di un numero maggiore di cifre di quelle che il calcolatore può visualizzare. Nei calcolatori viene usata per una ragione analoga. Abbiamo visto che 32 bit possono rappresentare un campo compreso approssimativamente tra −2000000000 e +2000000000, ma noi non vogliamo sempre numeri interi. Per esempio, in applicazioni contabili potremmo supporre di avere un campo decimale corrispondente alle due ultime cifre di destra perciò avremmo un campo di ± $ 20.000.000,00. Spesso viene fatto un qualche cosa di simile a quanto visto sopra, per cui la nostra rappresentazione usuale viene chiamata in *virgola fissa* (fixed point).

Con la virgola fissa esiste un problema importante: ogni cosa deve es-

sere predisposta in modo tale da accertare il caso più estremo. Per esempio, se si deve accettare 3,14159 la nostra virgola decimale si deve trovare, al minimo, spostata di cinque posizioni dall'estremità destra del numero, anche se nessun altro numero ha bisogno di più di due decimali. Fissando la virgola a cinque posizioni dalla destra si limita il campo rappresentabile a $\pm$ 20.000,00000. Questa soluzione è inadeguata per molte applicazioni.

La virgola mobile risolve questo problema *spezzando* il campo di numeri in molti campi sovrapposti con un numero di decimali variabili. Alcuni dei nostri 32 bit vengono utilizzati per indicare l'*esponente* (approssimativamente, la locazione della virgola decimale) ed alcuni per indicare la *mantissa* (approssimativamente, un numero in virgola fissa compreso nel campo indicato dall'esponente).

Per esempio, si supponga che la nostra parola di 32-bit venga divisa in due parti: una di 8 bit rappresentante l'esponente E in complemento a due ed una di 24-bit rappresentante, in complemento a due, la mantissa n. (Vedasi Figura 1.14). Il numero reale rappresentato da questo numero in virgola mobile è:

$$n \times 2^{E-23}$$

Questo è analogo alla notazione scientifica in cui si scrivono numeri nella forma $6{,}024 \times 10^{-23}$. Nella notazione scientifica si scrive $6{,}024 \times 10^{-23}$ al posto di espressioni equivalenti come 6024×10^{-26} o $0{,}00006024 \times 10^{-18}$. Analogamente c'è una forma normale per i numeri in virgola mobile: si usa il valore più grande possibile per n (che corrisponde a -23 in $n \times 2^{E-23}$). È facile vedere la ragione di tutto questo. Se nella nostra mantissa perdiamo posto per i primi zeri (come in $0{,}00006024 \times 10^{-18}$), si riduce lo spazio disponibile per ulteriori cifre significative. Per esempio, se ci fossimo limitati a 8 cifre di mantissa saremmo in grado di scrivere solamente $0{,}000060 \times 10^{-16}$.

Naturalmente questo è legato all'esponente. Normalizzando si potrebbe veramente spingere l'esponente fuori dal campo. (Per esempio, $6{,}024 \times 10^{-23}$ potrebbe non essere possibile se fossimo solamente in grado di avere esponenti fino a -20). In effetti, 8 bit forniscono esponenti compresi tra 2^{-128} e 2^{-127} che è migliore di $10^{-38} \div 10^{38}$. Molto raramente occorrono numeri di precisione o di ampiezze maggiori di queste. D'altra parte, una mantissa di 24 bit permette un campo approssimativo di $\pm$ 8 milioni, sei cifre significative, e spesso si ha bisogno di tutta questa estensione specialmente quando si accumulano somme di molti termini o si effettua la differenza di grandi numeri molto vicini l'uno all'altro.

Perciò i numeri in virgola mobile sono generalmente normalizzati. Pri-

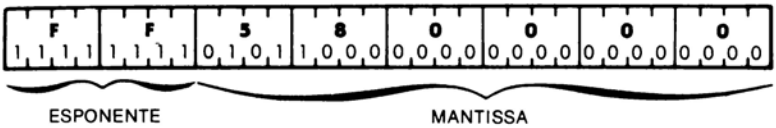


Figura 1.14 — Rappresentazione in Virgola Mobile di 0,34375

ma si è prestata maggior attenzione a che cosa dovrebbe succedere quando la normalizzazione spinge l'esponente al di sotto del valore minimo del campo possibile (*underflow*).

La Figura 1.14 illustra una rappresentazione in virgola mobile simile a quella discussa sopra. Il numero rappresentato è $0{,}34375_{10}$. La rappresentazione in virgola mobile è $FF580000_{16}$.

ESERCIZIO 14:

(a) Si spieghi perché FF580000 è la rappresentazione in virgola mobile di 0,34375. (Suggerimento: $0{,}34375 = 11 \times 2^{-5} = (11 \times 2^{19}) \times 2^{-1-23}$).

(b) Si scriva un algoritmo per sommare due numeri in virgola mobile. Lasciare il risultato nella forma normalizzata. (Suggerimento: $6.024 \times 10^{-23} + 1{,}236 \times 10^{-24} = 0{,}06024 + 1{,}236) \times 10^{-21}$).

(c) Si scriva un algoritmo per la moltiplicazione di due numeri in virgola mobile. Si lasci il risultato in forma normalizzata.

La virgola mobile è una rappresentazione utile e flessibile di un numero. Sfortunatamente, come per la rappresentazione BCD, lo Z8000 non possiede istruzioni che consentano di operare con numeri in virgola mobile.

Se si desidera usare la virgola mobile, occorre scrivere (ed avere) un *programma per la gestione della virgola mobile*. Nella maggior parte dei calcolatori questo comporta un lavoro non indifferente.

Rappresentazione di caratteri alfanumerici

Esiste una relazione ovvia tra la configurazione di bit di un byte o parola e i numeri che si desiderano rappresentare, ma non esiste niente di simile per i caratteri alfabetici. In realtà, non è esattamente chiaro che cosa occorre rappresentare: vogliamo i caratteri cinesi, gli accenti francesi, le lettere maiuscole e minuscole, diversi insiemi di caratteri, diversi colori?

Durante gli ultimi anni si è avuta una standardizzazione; i caratteri alfanumerici contraddistinti con codici «standard» sono le lettere maiuscole e minuscole dell'alfabeto inglese, i numeri $0 \div 9$, i simboli aritmetici (+, −, /, *, =, <, >), alcuni punti e simboli frequentemente usati come #, $, %, &. In tutto ci sono 90 caratteri alfanumerici.

Inoltre, spesso dobbiamo rappresentare i comandi di controllo macchina come il ritorno carello e cambio pagina come caratteri mescolati nel testo codificato. La trasmissione su linee telefoniche ha evidenziato il bisogno di *caratteri addizionali di controllo.*

Dal momento che ciascuna di queste cose deve essere rappresentata all'interno del calcolatore, la cosa più ovvia che viene in mente è di farla corrispondere ad un numero. Tale selezione e numerazione dei caratteri alfanumerici e di comando costituisce il *codice carattere.* Il principale codice utilizzato con i microcalcolatori è quello ASCII (American Standard Code for Information Interchange — Codice Standard Americano per lo Scambio di Informazioni). (Vedasi Figura 1.15).

Nell'ASCII ciascun carattere alfanumerico o di controllo è rappresentato da un numero compreso tra 0 e 127. Questo significa che occorrono 7 bit per rappresentare tutti i codici ASCII (siccome $2^7 = 128$). In realtà, per rappresentare ciascun carattere viene usato un byte di 8-bit. Nella rappresentazione interna l'ottavo bit (più significativo) è generalmente posto a zero; nella trasmissione tra la CPU e i dispositivi esterni, l'ottavo bit è spesso utilizzato per il controllo di errore. Uno di questi controlli di errore è la *parità.* Parità dispari, significa che nel carattere è presente un numero dispari di bit uno; mentre parità pari significa che il numero di bit posti a uno è pari. Il sistema utilizzato per il controllo di errore avviene indicando che tutti i caratteri trasmessi da o verso il calcolatore avranno, diciamo, parità dispari. Poi, in funzione del numero di bit ad uno, nel codice a 7-bit del carattere, l'ottavo bit è posto a uno o a zero prima di eseguire la trasmissione. Nello Z8000 è presente un'istruzione che controlla la parità di un byte ad 8-bit. Il bit P (che è in realtà lo stesso bit che rappresenta V) viene utilizzato come indicatore di parità. (Vedasi Figura 1.16).

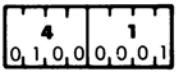
Essendo l'ASCII un codice a 7-bit, ciascun carattere ASCII è un numero esadecimale di due cifre compreso tra 00 e 7F. In Figura 1.15 è riportata la corrispondenza. Si noti che questi codici non sono assegnati in modo casuale. Piuttosto, le lettere maiuscole formano un blocco contiguo di codici (A = 41, B = 42, ... Z = 5A); analogamente le lettere minuscole (a = 61, b = 62, ... z = 7A) e i numeri (0 = 30, 1 = 31, ..., 9 = 39). Queste particolarità vengono usate in tutti i programmi.

ESERCIZIO 15: Si trovi la sequenza di caratteri ASCII che esprime «Hi! I'm a Z8000». (Ehi! Sono lo Z8000).
Non si dimentichino gli spazi. Quanti byte di memoria saranno occupati da questi caratteri?

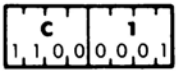
codice	carattere	codice	carattere	codice	carattere	codice	carattere
00	NUL	20[1]		40	@	60[5]	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27[2]	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	TAB	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C[3]	,	4C	L	6C	l
0D	CR	2D	—	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A		5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D[6]	}
1E	RS	3E	>	5E	↑	7E	~
1F	US	3F	?	5F[4]	←	7F[7]	RUBOUT

[1] spazio
[2] apice
[3] virgola
[4] o sottolineatura
[5] accento
[6] o ALT MODE
[7] DEL

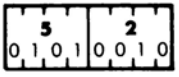
Figura 1.15 — Insieme dei Caratteri ASCII



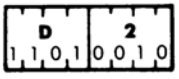
ASCII A (parità pari: 2 bit ad uno)



ASCII A (parità dispari: 3 bit ad uno)



ASCII R (parità dispari: 3 bit ad uno)



ASCII R (parità pari: 4 bit ad uno)

48 = ASCII H 69 = ASCII i

Rappresentazione ASCII di «Hi» con una parola di 16-bit (2 byte)

Figura 1.16 — Rappresentazione di un Testo ASCII

Capitolo II

Architettura Circuitale dello Z8000

In questo capitolo verrà descritta la struttura funzionale del microcalcolatore Z8000. Siccome la programmazione in linguaggio macchina è intimamente connessa con aspetti piuttosto dettagliati dell'architettura del calcolatore, questa descrizione è utile sia ai programmatori che ai progettisti circuitali.

Questo capitolo descrive, fondamentalmente, tutte le caratteristiche dello Z8000. Non esiste un sistema per presentare in un breve capitolo tutto questo materiale, in modo tale da essere immediatamente recepito dal lettore che non ha mai studiato prima d'ora l'architettura di un calcolatore. Tuttavia, dovreste fare ogni sforzo possibile per comprendere questo materiale, essendo esso la base per ciò che verrà descritto nel seguito del libro. Per esempio, se necessario, potreste leggere ripetutamente alcune parti, ma se dopo aver fatto tutto ciò vi accorgete che queste non vi sono ancora chiare, procedete tranquillamente; infatti, la maggior parte dei concetti essenziali alla comprensione dei programmi di questo libro sarà rispiegata in seguito, quando tali concetti saranno ripresentati.

L'architettura del microcalcolatore Z8000 è simile più a quella del PDP-11/34, ed altri famosi microcalcolatori, che a quella dei primi microcalcolatori del tipo Z80. Una macchina «general purpose» (di uso generale) così potente potrebbe risultare, in alcune situazioni, antieconomica; questo è il motivo per cui lo Z8000 viene fornito in due versioni: Z8001 e Z8002.

Tutta la potenza dello Z8000 è fornita dallo Z8001 unitamente al dispositivo, ad esso strettamente connesso, la MMU (Memory Management Unit – Unità di Gestione della Memoria). La CPU su un solo dispositivo Z8002 è funzionalmente equivalente allo Z8001, ma è limitata ad avere uno spazio molto più piccolo di memoria indirizzabile. La compatibilità verso l'alto è garantita da uno dei modi di funzionamento dello Z8001

nel quale può essere eseguita, senza essere modificata, la maggior parte dei programmi dello Z8002.

Lo Z8000 è un microprocessore a 16-bit, per cui i percorsi dei dati, le istruzioni, i registri, le operazioni aritmetiche e logiche sono tutti pensati per gestire parole di 16-bit. Molte delle sue istruzioni possono essere usate con *parole-lunghe* (32 bit), *byte* (8 bit) e *digit* (4 bit) ed è possibile indirizzare in modo indipendente, nel proprio spazio di memoria, ogni singolo byte.

Registri Non Specializzati

Lo Z8000 ha sedici *registri non specializzati*; questo significa che ciascun registro può essere utilizzato come *accumulatore, registro indice, registro base, registro indirizzo.* (In realtà, R0 può sempre essere utilizzato come accumulatore, ma esistono alcune limitazioni per quanto riguarda il suo utilizzo come registro indirizzo. Non può essere usato come registro indice o registro base).

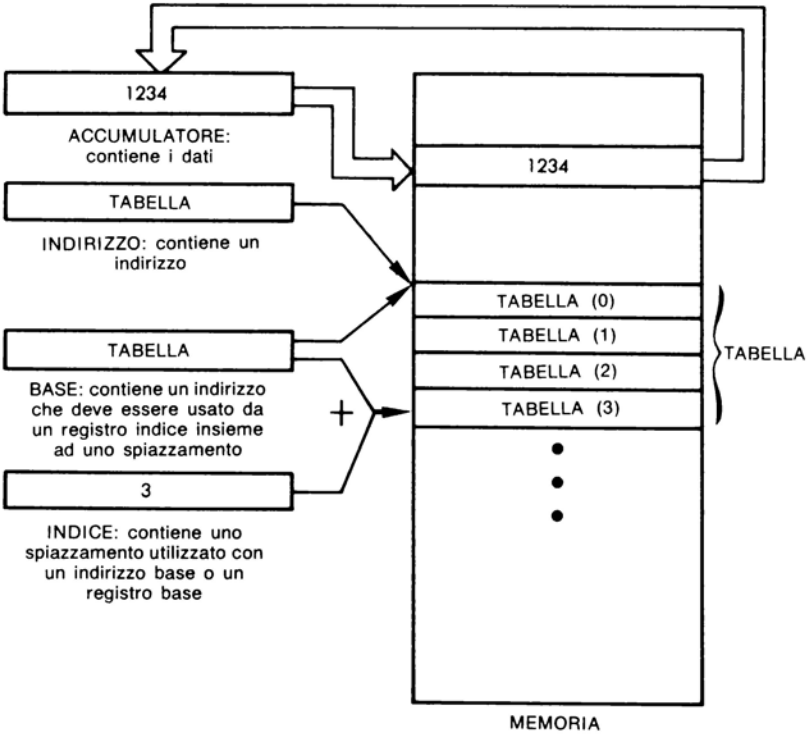


Figura 2.0 — Alcuni Usi dei Registri

Apriamo una parentesi per definire questi termini.

Un *registro* è un dispositivo elettronico in grado di memorizzare delle configurazioni di bit di una certa dimensione — in questo caso di 16 bit. Immaginatelo come una fila di 16 luci.

Un *accumulatore* è un registro utilizzato per contenere numeri (o configurazioni di bit non interpretate) mentre, o fino a quando, vengono eseguite su di esso operazioni aritmetiche o logiche.

Un *registro indice* è un registro che contiene la posizione riferita ad un elemento di una lista (chiamata anche tabella o insieme) — perciò se il registro contiene il valore 5, noi stiamo parlando riferendoci al quinto elemento di una qualche tabella. Quando si dirà: vogliamo un elemento della tabella «indicizzata dal» registro indice, siccome questo contiene il valore 5, ci fornirà il quinto elemento della tabella. L'indirizzamento indicizzato verrà discusso nel Capitolo V.

Un *registro indirizzo* è un registro che contiene l'indirizzo di memoria relativo all'elemento che il calcolatore sta per usare (per esempio, la coppia di registri H,L nello Z80). Nel Capitolo V si discuterà il modo di indirizzamento indiretto tramite registro.

Il registro base è simile al registro indirizzo dal momento che contiene un indirizzo di memoria, ma viene utilizzato insieme ad uno spiazzamento (offset) o indice, fornito in modo separato. Lo spiazzamento viene sommato all'indirizzo contenuto nel registro base per ottenere l'indirizzo dell'elemento che il calcolatore sta per usare. Nel Capitolo V verranno discussi i modi di indirizzamento riferiti al registro base e al registro base indicizzato.

I 16 registri non specializzati dello Z8000 possono essere suddivisi e combinati in diversi modi. I registri sono chiamati R0, R1, ..., R15. Essi possono essere combinati in otto coppie di registri: RR0, RR2, ..., RR14. R0 è la parte più significativa di RR0, mentre R1 è quella meno significativa, analogamente per gli altri. I registri possono essere ulteriormente combinati a gruppi di quattro: RQ0, RQ4, RQ8, RQ12; RR0 è la parte più significativa di RQ0 e RR2 è la parte meno significativa, ecc. Infine, i primi otto registri R0, R1, ..., R7 possono essere suddivisi in «H» e «L» (più significativo e meno significativo) per formare dei registri di dimensione pari al byte. RH0, RL0, RH1, RL1, ..., RH7, RL7. Tutto quanto detto sopra è illustrato in Figura 2.1.

Solamente 8 dei registri possono essere suddivisi in registri a byte; infatti se ci fossero più di 16 registri a byte, in una istruzione servirebbero più di 4 bit per indicare il registro a byte da usare. In un calcolatore con parole di istruzione di 16-bit, l'assegnazione di ciascun bit deve essere attentamente valutata in funzione dei potenziali benefici ricavabili; un gua-

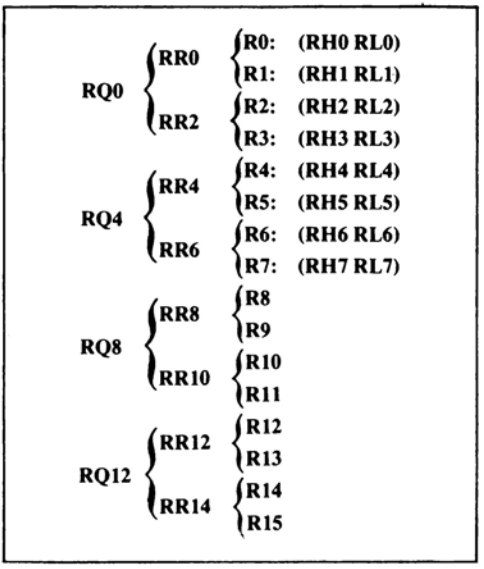


Figura 2.1. — Gerarchia dei Registri Non Specializzati dello Z8000

dagno di 16 registri a byte non compensa il prezzo di un ulteriore bit nel campo di identificazione del registro. Nel Capitolo IV verranno discussi in dettaglio i formati delle istruzioni.

Stack (Pila)

Le istruzioni ed i registri dello Z8000 sono stati pensati per facilitare l'uso dello *stack*. Lo stack può essere pensato come uno stack (pila) di piatti posti in una buca con una molla nel fondo, analogamente a quelli di una caffetteria. L'ultimo elemento posto nella pila è il primo ad essere rimosso, per cui lo stack viene chiamato buffer (LIFO) Last-in-First-Out (ultimo entrato — primo ad uscire).

Con la pila di piatti, quando vengono prelevati o aggiunti dei piatti, la cima è fissa mentre il fondo si muove su e giù. In un calcolatore è più facile muoversi nel modo opposto, per cui gli elementi, una volta che sono stati posti nella pila, non si muovono mai; piuttosto lo *stack pointer* (puntatore della pila) punta sempre alla locazione di memoria che contiene l'ultimo elemento aggiunto alla pila.

Per adesso, si pensi alla memoria dello Z8000 come ad una sequenza di registri di 16-bit ciascuno dei quali viene identificato da un numero (chiamato indirizzo). Il puntatore dello stack è un registro indirizzo che

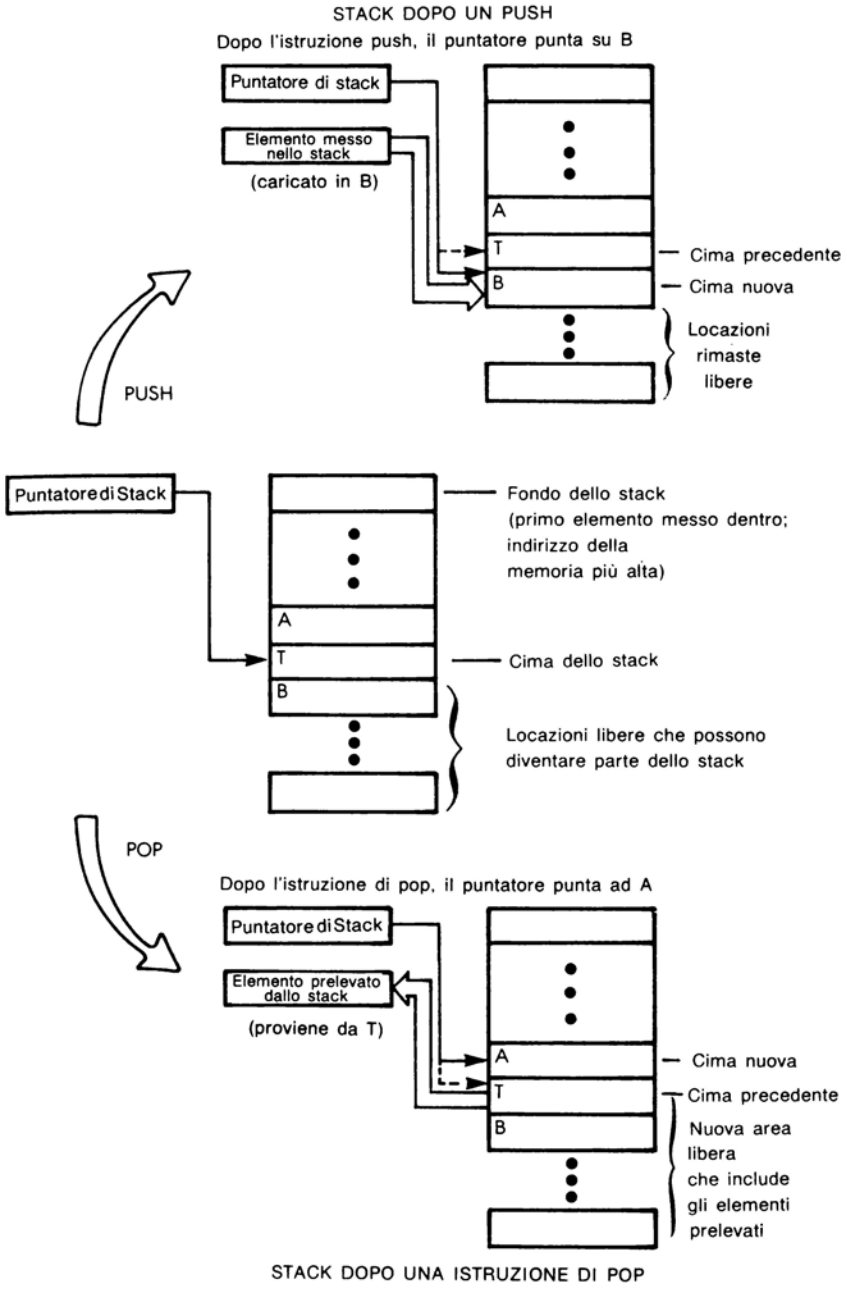

STACK DOPO UN PUSH
Dopo l'istruzione push, il puntatore punta su B
Puntatore di stack
Elemento messo nello stack
(caricato in B)
A
T
B
Cima precedente
Cima nuova
Locazioni rimaste libere
PUSH
Puntatore di Stack
Fondo dello stack (primo elemento messo dentro; indirizzo della memoria più alta)
Cima dello stack
Locazioni libere che possono diventare parte dello stack
POP
Dopo l'istruzione di pop, il puntatore punta ad A
Puntatore di Stack
Elemento prelevato dallo stack
(proviene da T)
Cima nuova
Cima precedente
Nuova area libera che include gli elementi prelevati
STACK DOPO UNA ISTRUZIONE DI POP

Fig. 2.2 — Stack

punta alla locazione di memoria corrispondente in quel momento alla cima dello stack (vedasi Figura 2.2). Quando qualche cosa viene *spinto* (PUSH) nello stack (aggiunto), questo viene posto nella locazione successiva alla cima della pila in quel momento, e il puntatore dello stack viene cambiato, in modo da puntare alla posizione della nuova cima. Quando viene *rimosso* (POP) un elemento dalla pila, la posizione che esso occupava diviene libera e il puntatore viene cambiato in modo da posizionarsi sull'elemento che si trova sotto a quello appena estratto che diviene la nuova cima.

Ciascun registro indirizzo dello Z8000 può essere usato come puntatore di stack; inoltre ci sono delle istruzioni dello Z8000 utilizzabili per rimuovere o aggiungere elementi allo stack. Casualmente, come riporta la Figura 2.2, il fondo dello stack ha l'indirizzo più alto ed ogni elemento spinto dentro lo stack riceve il successivo indirizzo più basso rispetto al precedente valore della cima. Giustamente, si sarebbe potuto fare facilmente nell'altro modo, ma questo schema permette di suddividere facilmente la memoria tra stack e dati o programmi: iniziandoli agli estremi opposti della memoria e facendoli crescere uno verso l'altro; oppure iniziandoli schiena a schiena e facendoli crescere in direzione opposta.

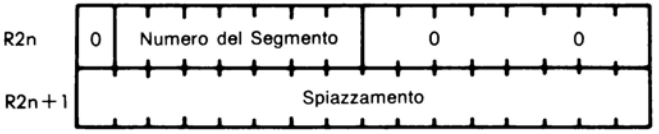
Lo Z8000, per alcune operazioni di stack che si generano automaticamente, definisce come puntatore di stack un particolare registro indirizzo che memorizza il ritorno da sottoprogrammi (subroutine) oppure lo stato di un processo quando vengono riconosciute delle interruzioni o delle trappole (trap). Più avanti in questo capitolo si analizzeranno le interruzioni e le trappole mentre le subroutine verranno analizzate nel Capitolo III. Il registro definito per le operazioni di stack dipende dal funzionamento nel modo segmentato o non segmentato dello Z8000. Adesso spiegheremo che cosa significa.

Segmentazione della Memoria

Lo Z8000 permette l'indirizzamento del singolo byte: cioè ciascun byte di memoria è identificato da un proprio indirizzo. I registri a 16-bit, R1, R2, ..., R15 possono individuare solamente 2^{16} indirizzi diversi (si ricordi la nostra regola: n bit possono distinguere 2^n elementi). Perciò l'estensione di indirizzi dello Z8000 potrebbe contenere 65.536 byte. In effetti, l'architettura dello Z8000 permette di gestire 128 estensioni diverse di questi 65.536 byte.

Esistono due diversi modi di operare. In uno, chiamato *non-segmentato*, la CPU Z8000 esegue i programmi che si trovano in uno di questi campi di 65.536 byte. Tutti gli indirizzi sono di 16 bit e il programma non possiede un modo esplicito per influenzare in qualche modo un altro cam-

po. Nell'altro modo di funzionamento, chiamato *segmentato*, la CPU Z8000 riconosce gli indirizzi in un formato di due-parole che permette l'indirizzamento diretto di un qualsiasi byte in uno qualsiasi dei 128 campi di indirizzi. La Figura 2.3 illustra come, questo formato indirizzo a due parole, appaia in uno qualsiasi dei registri a parola lunga RR2, RR4, ..., RR14 che possono essere utilizzati nel modo *segmentato* come registri indirizzo.



(n = 1,. . .,7)

Figura 2.3 — Formato del Registro per gli Indirizzi Segmentati

Nel paragrafo relativo allo stack si è detto che un registro indirizzo veniva definito come il *puntatore implicito dello stack*, questo è il registro usato per memorizzare automaticamente i ritorni dalla subroutine o dagli stati dei processi. Nel modo non segmentato, questo registro, utilizzato come puntatore di stack è R15; nel modo segmentato è RR14.

A dispetto del fatto che nel modo segmentato si usano indirizzi di 23-bit e nel modo non-segmentato indirizzi di 16-bit, le istruzioni per i due modi sono molto simili. Questo dipende dal fatto che nell'architettura dello Z8000, gli indirizzi si ritrovano solo in due luoghi: nei registri indirizzo e come parte delle istruzioni. Le istruzioni che usano i registri indirizzo per specificare i loro argomenti hanno un formato identico per entrambi i modi. Quando un indirizzo è contenuto in una istruzione, esso è sempre posto alla fine dell'istruzione base. Nel modo non-segmentato viene aggiunto un indirizzo di una parola; nel modo segmentato l'indirizzo viene aggiunto in uno dei due possibili formati; in entrambi i casi si aggiunge l'indirizzo alla stessa istruzione base.

Nel caricamento di indirizzi nei registri i due modi potrebbero essere espressi mediante istruzioni diverse, però in questo caso viene fornita una istruzione speciale (LDA) per caricare in un registro indirizzo l'indirizzo del proprio argomento. Questa istruzione ha la stessa forma in entrambi i modi.

Mappa della Memoria

Fino ad ora questa suddivisione della memoria dello Z8000 in 128 campi di indirizzi ci è sembrata un modo per avere una macchina dotata

di una grossa capacità di memoria, ma con istruzioni relativamente brevi e con registri indirizzi piccoli. In realtà, può darsi che questa fosse l'idea originale, ma è anche vero che introducendo il concetto di *mappatura di memoria* si evidenziano ulteriormente la potenza e la flessibilità dello Z8000.

La mappatura di memoria si basa sul seguente modo di operare: il segmento di indirizzo di 23-bit è trattato come un indirizzo *logico*. Per mezzo di un calcolo si perviene al reale indirizzo fisico a cui esso corrisponde. Il numero del segmento, a 7-bit, viene tradotto (mappato) in un indirizzo nella memoria fisica reale che diviene il punto di inizio del campo di indirizzi di 65.536 byte di quel segmento. (Vedasi Figura 2.5). Partendo da questa idea di base, si possono sviluppare molte varianti:

- Non tutti i segmenti devono avere la stessa dimensione; alcuni possono essere più piccoli di 65.536 byte. (Vedasi Figura 2.4)

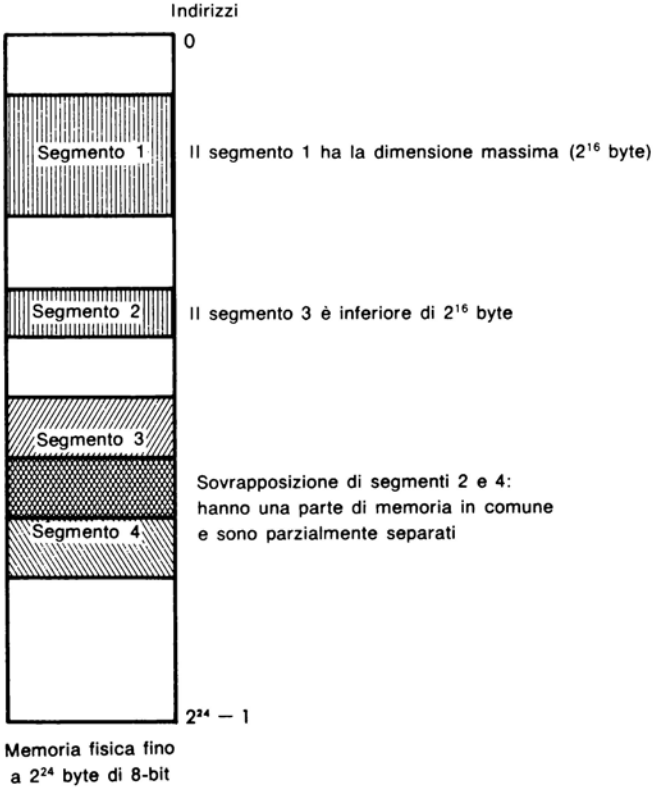


Figura 2.4 — Alcune Caratteritiche della Mappatura di Memoria

- Nella memoria fisica (Vedasi Figura 2.4) i segmenti possono avere delle aree di sovrapposizione.
- È possibile memorizzare determinati attributi dei segmenti e individuare le operazioni incompatibili con questi attributi (esempio impedire la scrittura in un segmento definito di sola-lettura).
- La locazione e gli attributi dei segmenti possono essere cambiati in modo dinamico.

Tali elaborazioni richiedono un supporto logico, che nell'architettura dello Z8000 è fornito da una unità esterna che gestisce la *mappatura della memoria* (MMU - Memory Management Unit).

Se la MMU lavorasse come è stato descritto prima, dovrebbe ricevere in ingresso l'indirizzo a 23-bit, inviato dall'uscita dello Z8000, convertire la porzione a 7-bit che esprime il numero del segmento nell'indirizzo fisico di base, sommare lo spiazzamento, costituito dai rimanenti 16-bit, all'indirizzo di base e inviare sulla propria uscita l'indirizzo fisico risultante. Dal momento che l'architettura del sistema dello Z8000 è basata su un indirizzo fisico costituito da 24-bit, questo significa che la MMU dovrebbe avere i registri interni e l'unità aritmetica in grado di operare con 24-bit e al limite 23 piedini per l'ingresso e 24 per l'uscita dell'indirizzo.

Onde ridurre la complessità della gestione, la memoria fisica viene mappata solamente in blocchi di 256 byte. Gli 8-bit meno significativi, dello spiazzamento di 16-bit, arrivano direttamente alla memoria fisica. La MMU converte il numero del segmento, espresso da 7-bit, nei 16-bit più significativi dell'indirizzo fisico di base a 24-bit; poi somma gli otto bit più significativi dello spiazzamento a questo valore per ottenere i 16-bit più significativi del reale indirizzo fisico della memoria.

Questa procedura è illustrata in Figura 2.5. Lo schema adottato comporta una lieve perdita di flessibilità ma permette di conseguire una notevole semplificazione della MMU.

Per ora, non vogliamo andare nel dettaglio di come venga specificata alla MMU l'informazione riguardante la locazione e gli attributi del segmento. Questa informazione è trasmessa dallo Z8000 come se fosse inviata ad una periferica di I/O ma dal momento che la MMU non usa le linee di indirizzo da 7 a 0, tutte le informazioni transitano sopra le linee 15-8. Nei Capitoli IV e VII parleremo della speciale classe di istruzioni di I/O che permettono di trattare questo problema.

Una MMU può gestire fino a 64 segmenti; due MMU sono sufficienti a gestire l'intero campo di 8.388.608 byte indirizzabile dallo Z8000.

Spazi Multipli di Indirizzi

Gli 8.388.608 byte indirizzabili dallo Z8000 sono definiti come lo *spa-*

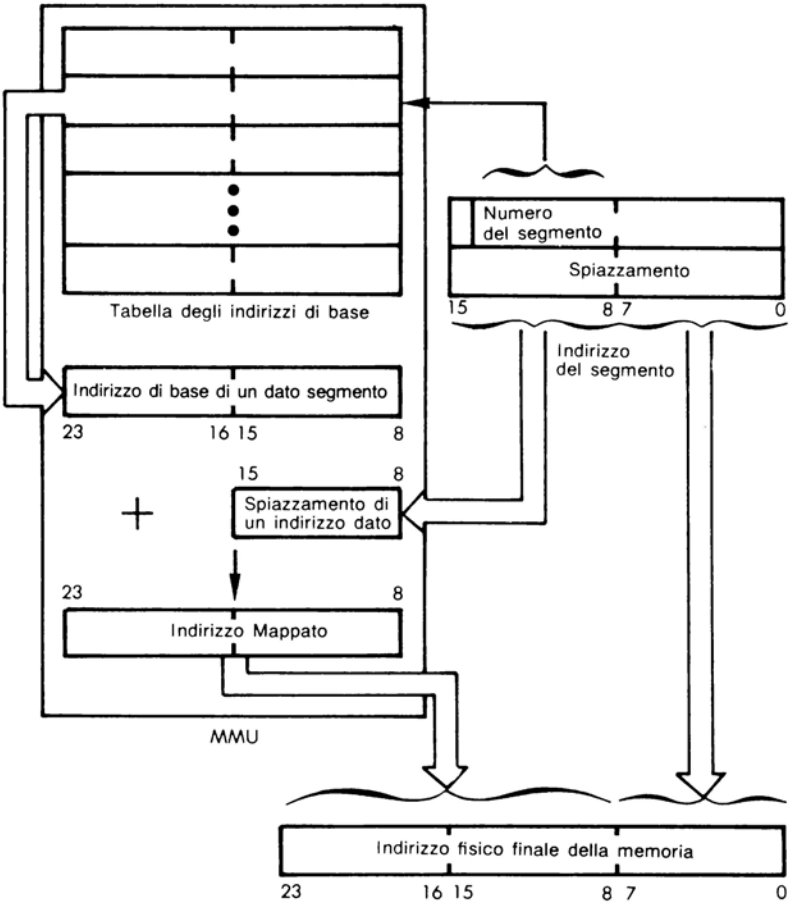


Figura 2.5 — Calcolo dell'Indirizzo Mappato

zio indirizzo dello Z8000, ma in effetti le specifiche dello Z8000 indicano che è possibile avere un massimo di sei spazi di indirizzi di questo tipo. Questo viene realizzato fornendo informazioni di stato addizionali che permettono a della logica esterna di selezionare diverse MMU (o diverse memorie fisiche) in funzione dei diversi tipi di accesso alla memoria (esempio estrazione di istruzioni usata come l'opposto di un accesso all'area dei dati).

I tre diversi tipi di accesso alla memoria, distinguibili tramite l'informazione di stato, sono: accesso alla stack; accesso all'area istruzioni, e accesso all'area dati. Inoltre (si veda più avanti) esiste una linea di stato che permette di distinguere il modo sistema da quello normale. In definitiva, si possono distinguere sei spazi separati di indirizzi.

Da tutto questo ne consegue che, oltre alla protezione totale fornita dalla separazione completa di questi spazi, viene notevolmente aumentata la capacità della memoria dello Z8000; questo è particolarmente importante per lo Z8002 limitato ad un campo di 65.536 byte indirizzabili separatamente.

D'altra parte, vedremo nei Capitoli IV e V che ogni tentativo di separazione degli spazi degli indirizzi di questo tipo può comportare delle difficoltà.

Modi Sistema e Normale

Dal momento che l'architettura dello Z8000 permette di avere grosse configurazioni, possibilmente con vari programmi eseguiti in modo concorrente, occorre un espediente per evitare che i programmi difettosi non interferiscano con gli altri. Inoltre, l'organizzazione di un buon programma richiede che certe funzioni chiave siano centralizzate e con accesso controllato. Queste idee, e varianti di esse, hanno portato nei primi grossi sistemi di calcolo al concetto di *istruzioni privilegiate*: esiste cioè una modalità: di funzionamento definita come *modo sistema*, nella quale possono essere eseguite tutte le istruzioni di cui è dotato il calcolatore; ne esiste un'altra definita come *modo normale* nella quale non possono essere eseguite le istruzioni privilegiate.

I modi sistema e normale usano puntatori diversi per lo stack. In entrambi i modi il registro viene identificato come R15 (o RR14 se opera nel modo segmentato). Non esiste alcun espediente per far si che un programma che gira in modo normale abbia accesso al puntatore di stack del modo sistema, ma nel modo sistema si può accedere al puntatore dello stack di modo normale considerandolo come un *registro di controllo* identificato con NSP. Parleremo dei registri di controllo un po' più avanti in questo capitolo.

Lo Z8001 e lo Z8002

Fino ad ora abbiamo parlato dello Z8000 come di un unico microprocessore caratterizzato da due modalità di funzionamento: segmentato e non-segmentato. In realtà stiamo descrivendo lo Z8001. C'è una versione ridotta chiamata Z8002, che non può funzionare nel modo segmentato e che ha un campo di indirizzamento di 65.536 byte. La maggior parte dei programmi scritti per lo Z8002 può essere eseguita in uno dei segmenti dello Z8001. Il dispostivo Z8002 ha esattamente, a livello di piedini, le stesse funzioni dello Z8001, escluse le sette linee che definiscono il nume-

ro del segmento ed una linea (trappola di segmento) di ingresso dello Z8001 proveniente dalla MMU.

Dal momento che questo è un libro sulla programmazione dello Z8000, e siccome tutti i programmi dello Z8002 possono girare nello Z8001, non c'è bisogno di parlare molto dello Z8002; tuttavia, ci sono alcuni casi in cui lo Z8001 in versione non-segmentata differisce dallo Z8002; questi casi verranno evidenziati mano a mano che si incontreranno.

Stato della CPU

Abbiamo parlato dello spazio indirizzo del programma dello Z8000 e della maniera con cui le istruzioni memorizzate vengono eseguite: come se fossero dei passi di un algoritmo. La CPU tramite il *program counter* (PC) (contatore di programma) si ricorda del punto in cui si trova nello spazio indirizzi in cui è allocato il programma (esempio si ricorda quale passo del programma sta eseguendo). Il PC contiene sempre l'indirizzo, riferito allo spazio di memoria ove è allocato il programma, al quale si deve estrarre la prossima istruzione. Ogni volta che viene estratta una istruzione, il PC viene lasciato in modo da indicare la posizione successiva. La sequenza con cui viene eseguito un programma è simile alla sequenza dei simboli rettangolari di un diagramma di flusso. Le istruzioni corrispondenti ai simboli romboidali cambiano il contenuto del PC.

Il PC è un registro indirizzo, con funzione-dedicata, che determina la metà di quello che è chiamato lo stato della CPU. L'altra metà è la *Flag / Control Word* — Parola di Controllo/Indicatore (FCW). (Vedasi Figura 2.6). Le istruzioni corrispondenti ai simboli romboidali del diagramma di flusso dipendono dal byte meno significativo del FCW, chiamato FLAGS (INDICATORI). FLAGS è il gruppo di bit di cui si è brevemente discusso nel Capitolo I: C, Z, S, V (conosciuto anche come P), D e H. Nel Capitolo IV, descrivendo l'insieme delle istruzioni dello Z8000, verranno indicate le istruzioni che agiscono sullo stato di questi bit; nello stesso capitolo verrà anche illustrato il modo in cui queste istruzioni agiscono.

Figura 2.6 — Identificazione del significato dei Bit nella Parola degli Indicatori di Controllo (FCW)

Per il momento indichiamo qui di seguito il modo in cui sono generalmente usati:

C: Riporto aritmetico da byte, parola o parola lunga.
Z: L'ultima operazione ha dato risultato zero.
S: L'ultima operazione ha lasciato il bit più significativo (segno) uguale ad uno; esempio, quando il risultato è negativo.
V: Supero aritmetico (esempio, la somma di due numeri positivi è negativa).
P: Il byte controllato ha parità pari.
D, H: Usato dall'istruzione DAB dopo la somma o la sottrazione di un byte.

L'altra metà di FCW contiene quattro bit di controllo: due bit per il modo di funzionamento e due bit per l'abilitazione dell'interruzione. Parleremo delle interruzioni più avanti in questo capitolo; per adesso, diciamo che questi bit di abilitazione controllano se vengono servite le interruzioni ad essi corrispondenti o se invece vengono lasciate in attesa. I bit di controllo di FCW sono:

SEG: È uguale ad uno per il funzionamento nel modo segmentato, a zero per quello non-segmentato.
$S/\overline{N}$: È uguale ad uno quando funziona in modo sistema: a zero per il modo normale.
VIE: Viene posto ad uno per abilitare le interruzioni vettoriali, a zero per mascherarle.
NVIE: Viene posto ad uno per interruzioni non-vettoriali, a zero per mascherarle.

Il PC dello Z8001 è un registro indirizzo costituito da una parola con il formato rappresentato in Figura 2.3. Quando lo Z8001 lavora in modo non-segmentato, la parte del PC che identifica il numero del segmento non cambia, ma viene ugualmente inviata sui piedini d'uscita che definiscono il numero del segmento. Nello Z8002 il PC è costituito da un registro a 16-bit; non esistono i piedini che identificano il numero del segmento.

La FCW, d'altra parte, è sempre la stessa. È un registro dedicato di 16-bit. L'unione del PC e della FCW costituisce lo *stato della CPU*, ma ci sono varie complicazioni.

In due casi, lo stato della CPU viene considerato come una entità unica: quando al verificarsi di interruzioni esso viene salvato nello stack (si veda il prossimo paragrafo), e quando, tramite l'istruzione LDPS, può es-

sere ricaricato dalla memoria. L'istruzione LDPS riconosce un formato per il funzionamento segmentato ed un altro per quello non-segmentato. Nella forma non-segmentata è costituito da una parola lunga con FCW posto nella parola più significativa e PC in quella meno significativa; nella forma segmentata è costituito da quattro parole: uno 0, la FCW, e due parole per il PC. D'altra parte, indipendentemente dal modo di segmentazione, lo stato della CPU è memorizzato nello stack in un certo formato per lo Z8002 ed in un altro per lo Z8001. Nello Z8002 è espresso dalla parola lunga (FCW, PC) uguale alla versione di LDPS per il modo non-segmentato, ma nello Z8001, è espresso da tre parole: FCW seguita dalla doppia parola di PC. Tutto questo viene mostrato in Figura 2.7.

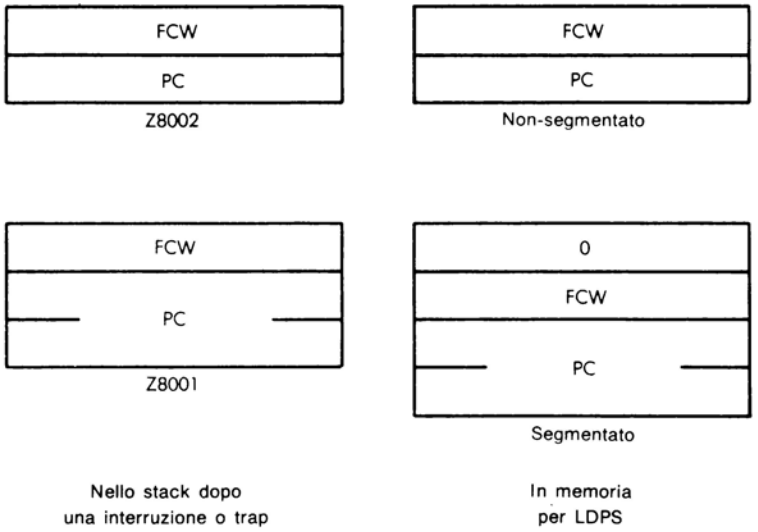


Figura 2.7 — Stato della CPU per Interruzioni e LDPS

Questo è un importante punto di incompatibilità tra lo Z8002 e lo Z8001 nel funzionamento non-segmentato; le conseguenze di questo verranno discusse in maggior dettaglio nel Capitolo IX quando parleremo di parametri trasmessi alle routine, invocate da una chiamata al sistema, da parte dell'individuazione di una trappola.

Trappole e Interruzioni

Tutti abbiamo provato cosa significhi essere interrotti durante lo svolgimento di una attività e cercare di ritornare alle condizioni precedenti all'interruzione una volta che abbiamo finito di gestirla.

Le interruzioni sono fastidiose, ma un aspetto importante dell'intelligenza umana è la capacità di gestirle. Le nostre capacità sarebbero fortemente sminuite se dovessimo sempre finire una cosa prima di iniziarne un'altra o se dovessimo sempre far ripartire dall'inizio il lavoro interrotto.

Le interruzioni possono provenire da una fonte esterna — o qualcuno o qualcosa altro — oppure possiamo interrompere noi stessi, come quando pensiamo improvvisamente a qualche cosa di più importante di quello che stiamo facendo.

Questo è simile a quello che si verifica in un calcolatore. Quando «qualcun altro» interrompe, si parla di *interruzione*; quando il calcolatore si autointerrompe si parla di *trappola* (trap).

Che tipo di gestione possiamo usare per fare in modo di poter ripartire da dove eravamo rimasti dopo aver eseguito l'interruzione? Possiamo mettere un segnalibro nel libro o ricordarci «mentalmente» dove eravamo rimasti. Qualche cosa che ricordi lo stato di ciò che stavamo facendo in modo sufficientemente dettagliato da permetterci di ritornarci.

Lo Z8000 fa la stessa cosa: quando viene interrotto salva lo stato della CPU nello stack di *sistema*; dopo aver servito l'interruzione ristabilisce lo stato della CPU estraendo e riportando in FCW e PC il valore precedentemente salvato.

Oltre a questo metodo usato per ricordarsi il proprio stato, lo Z8000 ha bisogno di un mezzo per dire che cosa ha causato l'interruzione o la trappola. Questo è realizzato con una parola (chiamata *ragione*) che viene salvata nello stack dopo lo stato della CPU; la ragione viene fornita dal dispositivo esterno che ha causato l'interruzione, o dalla CPU se si tratta di una trappola. (Vedasi Figura 2.8).

Una interruzione è generalmente associata ad un dispositivo di I/O. Per esempio, la stampante può essere pronta per ricevere un altro carattere, qualcuno può aver premuto un tasto sulla tastiera del videoterminale, oppure uno qualsiasi degli altri dispositivi potrebbe richiedere l'attenzione della CPU. Una trappola è causata dall'esecuzione (o tentativo di esecuzione) di una istruzione. Una *chiamata del sistema* mediante trappola è un mezzo che consente ad una routine eseguita in modo normale di richiamare una delle 256 routine indipendenti di sistema. Tutte le altre interruzioni da trappole segnalano errori di condizione, come per esempio l'utilizzo di una istruzione non implementata, l'utilizzo di istruzioni privilegiate quando si opera in modo normale, il tentativo di accedere ad un indirizzo di segmento in conflitto con le tabelle di attributo del segmento della MMU. La trattazione del funzionamento della trappola verrà discussa nel Capitolo IX.

Per le trappole, la ragione memorizzata nello stack è la prima parola

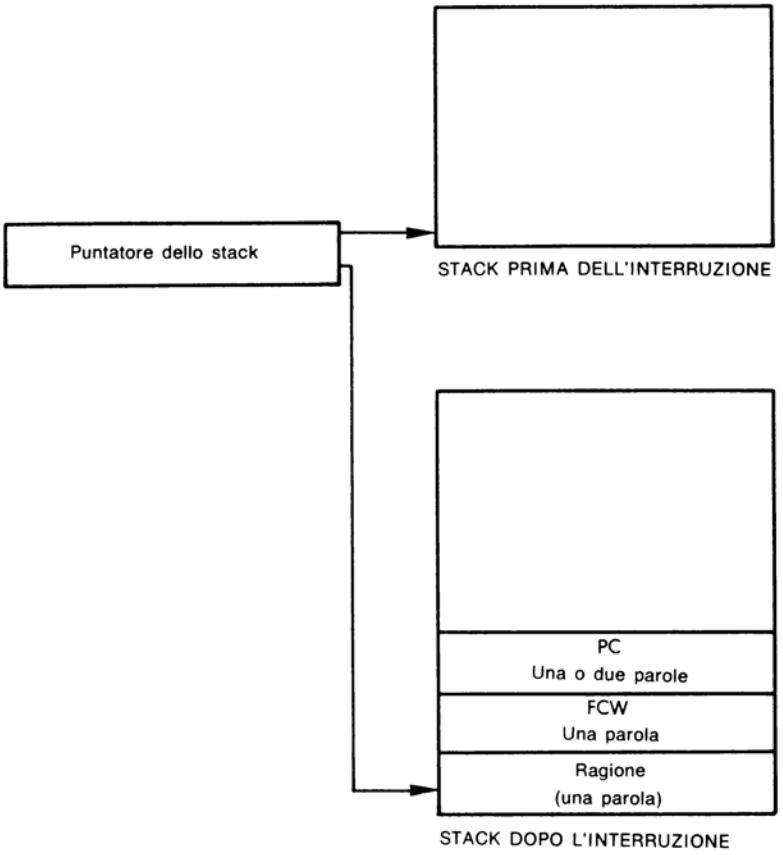


Figura 2.8 — Stack dopo una Interruzione

dell'istruzione che origina la trappola, tranne la trappola causata da segmentazione — per questa l'identificazione della MMU si trova nel byte più significativo e niente di interessante in quello meno significativo. I 16-bit che costituiscono la ragione sono forniti dai dispositivi che generano interruzioni per ogni informazione di stato che possa essere utile. L'unica richiesta è che nell'interruzione vettoriale il byte meno significativo della ragione sia un indice ad una tabella di 256-parole di indirizzo di routine di gestione dell'interruzione.

Questa tabella indirizzi fà parte del *program status area* (PSA — area di stato del programma), un'area definita dall'utente nella memoria *programma* ed indirizzata da un registro di controllo chiamato *program status area pointer* (PSAP — puntatore dell'area di stato di programma). La

Figura 2.9 riporta la configurazione dell'area di stato di un programma. Gli indirizzi forniti sono riferiti all'indirizzo di partenza specificato in PSAP. Casualmente, gli ultimi 8-bit di PSAP sono forzati a zero, in modo tale che la PSA inizi sempre in coincidenza con l'inizio del blocco fisico della memoria. In funzione dei due diversi formati dello stato della CPU nella memoria (vedasi Figura 2.7) il fattore *s* assume il valore 1 o 2. La lunghezza reale di PSA è determinata dal massimo indice possibile dell'interruzione vettoriale.

Quando si riscontra una interruzione o una trappola, lo stato della CPU viene messo nello stack di sistema, seguito dalla ragione: poi viene caricato il nuovo stato della CPU dal blocco di stato del programma corrispondente alla particolare trappola o interruzione. Caricando il nuovo stato della CPU si trasferisce il controllo di questa alla routine fornita dall'utente per gestire l'interruzione o la trappola verificatasi (siccome il valore del PC è parte dello stato). Quando finisce la routine, viene eseguita una istruzione di ritorno da interruzione che permette di estrarre dallo stack di sistema la ragione e lo stato della CPU precedentemente salvato. La ragione viene scartata mentre viene ristabilito lo stato precedentemente salvato. In questo modo lo Z8000 ritorna a fare ciò che stava facendo prima di essere interrotto.

Escluse situazioni molto particolari, la routine di gestione dell'interruzione non deve alterare con le sue operazioni i registri non specializzati, dal momento che si può verificare in ogni istante una interruzione, indipendentemente da ciò su cui sta lavorando il processore. Se si lasciasse un registro cambiato, questo apparirebbe al programma interrotto come se si fosse cambiato da solo tra due istruzioni consecutive e questo renderebbe impossibile la normale programmazione. Per cui la routine che gestisce l'interruzione deve salvare e successivamente ristabilire ogni registro che intende usare. Questo argomento verrà trattato nel Capitolo IX.

I tre tipi di interruzioni possibili sono: interruzione non-mascherabile (NMI), interruzione non-vettoriale (NVI) e interruzione vettoriale (VI). Il tipo NMI è generalmente riservato a qualche cosa che non può aspettare, come per esempio una segnalazione di mancanza di alimentazione, che non può essere fermata. Le NVI e VI sono interruzioni mascherabili. Questo significa che possono aspettare fino a quando in FCW i corrispondenti bit di abilitazione NVIE e VIE vengono posti ad uno. Il NVI si basa sul programma per determinare che cosa lo ha causato; esso richiede una logica circuitale più semplice di quella utilizzata per VI, siccome non occorre mettere un indice negli 8-bit meno significativi della ragione. Per porre un indice negli 8-bit meno significativi della ragione, il VI utilizza il dispostivo che ha causato l'interruzione, poi la CPU usa, automati-

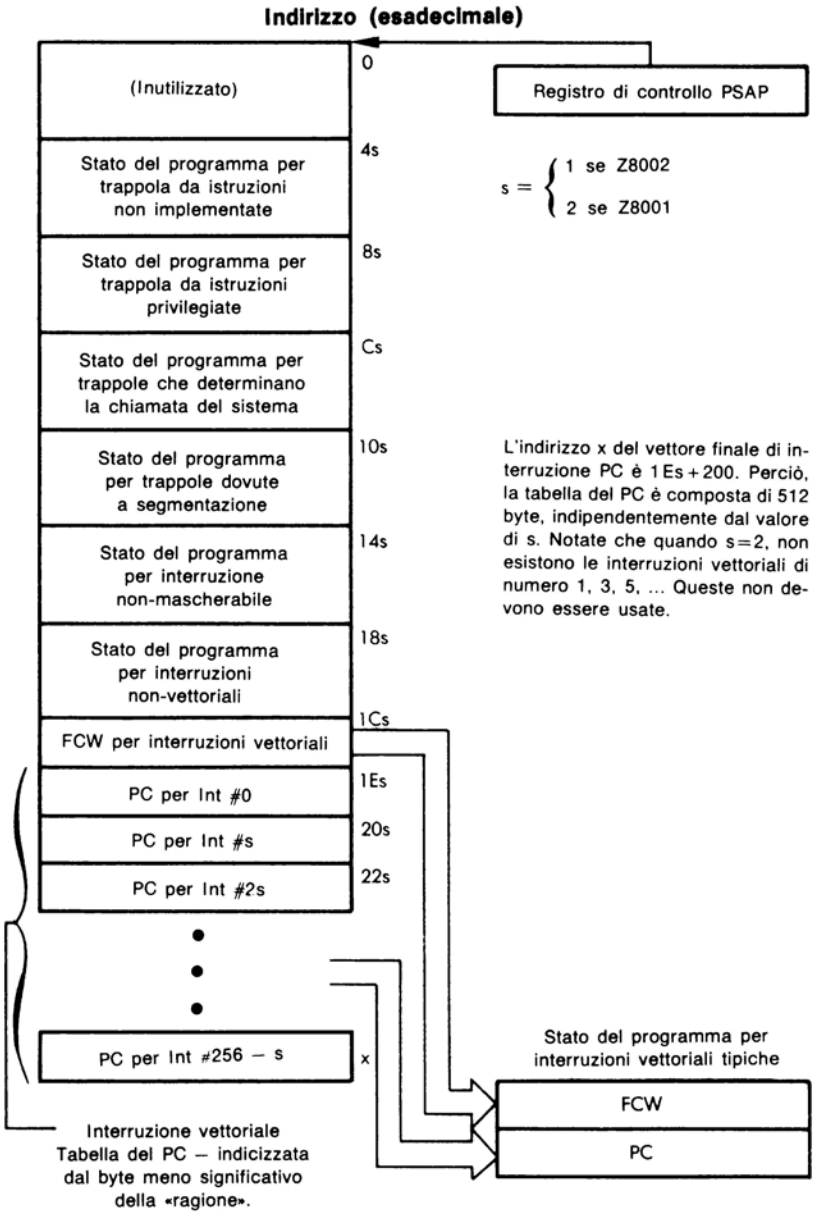


Figura 2.9 — Stato del Programma per le Interruzioni e le Trappole

camente questo indice per estrarre dalla tabella dei vettori di interruzione uno dei valori del PC. Per lo Z8001 l'indice *deve essere un numero pari* (0, 2, 4, ..., 254). (Vedasi Figura 2.9).

Registri di Controllo

Durante la discussione precedente abbiamo incontrato diversi registri di controllo. Fondamentalmente ce ne sono quattro: FCW, PSAP, NSP e REFRESH. Il byte meno significativo di FCW è chiamato FLAGS e può essere analizzato separatamente. PSAP e NSP sono registri indirizzi. Nella versione segmentata essi hanno parti separate, per il numero del segmento e lo spiazzamento, chiamate rispettivamente PSAPSEG, PSAPOFF, NSPSEG, NSPOFF e dal momento che non ci sono parole doppie per il trasferimento da o verso uno di questi registri, essi debbono essere chiamati con questi nomi. Questo è un punto importante:

ATTENZIONE: Siccome la versione della doppia parola di PSAP deve essere cambiata in due passi successivi, occorre prevedere delle protezioni per evitare delle interruzioni *durante* l'esecuzione dei due passi.

Se, per esempio, si verifica una interruzione mentre PSAP è per metà una cosa e metà un'altra, lo stato intermedio potrebbe corrispondere all'indirizzo di un PSA valido, cosa possibile ma molto improbabile.

L'ultimo registro di controllo è chiamato REFRESH. Questa caratteristica dell'architettura dello Z8000 può sembrare inconsueta agli utilizzatori di minicalcolatori, ma il concetto è molto semplice. Una *memoria dinamica* è una RAM economica che può ricordare le cose solo per alcuni millisecondi. Ciò che la rende utile è il *circuito di rinfresco*, un meccanismo che permette di passare attraverso le locazioni della memoria dinamica leggerne i contenuti, e riscriverli immediatamente in essa.

Nello Z8000 il registro di REFRESH fornisce il supporto necessario al circuito di rinfresco. Esso memorizza la prossima *riga* di memoria da rinfrescare (una specie di PC per il circuito di rinfresco), e controlla la frequenza del *ciclo di rinfresco*. Il registro REFRESH è costituito da un bit di abilitazione (RE), nel campo RATE (durata ciclo) di 6-bit e da un campo per il conteggio della riga (ROW) di 9-bit (vedasi Figura 2.10).

Le operazioni dello Z8000 sono sincronizzate con un ingresso di clock che può avere un ciclo veloce di 250 ns (impulso di clock) od uno lento di 2 μ s. Il campo del RATE determina la frequenza con cui lo Z8000 esegue una pausa per un ciclo di rinfresco. Il valore 1 significa una volta ogni 4 cicli di clock, 2 significa ogni 8 cicli, 3 significa ogni 12, ecc.. Un ciclo di rinfresco ha una durata pari a tre cicli di clock. Per confronto, le istruzio-

ni Z8000 possono essere eseguite dalla CPU in un tempo che varia di volta in volta da 3 a 10 cicli di clock. (Casualmente, il temporizzatore associato al registro REFRESH continua sempre ad andare avanti, per cui se un ciclo di rinfresco viene ritardato mentre aspetta la fine dell'esecuzione di una istruzione, il temporizzatore farà partire il prossimo ciclo di rinfresco esattamente come se quello precedente fosse partito in tempo).

Durante l'esecuzione del rinfresco i contenuti del campo a 9-bit, ROW, vengono inviati al circuito di rinfresco; poi ROW viene incrementato di due. Perciò, il campo ROW è in realtà un contatore con parola ad 8-bit che viene visto come un contatore con parola di 9-bit. Il suo bit meno significativo è sempre a zero.

Quando la CPU si trova nello stato di STOP, esegue continuamente dei cicli di rinfresco, permettendo in tal modo di utilizzare un un circuito esterno di rinfresco. Quando la CPU viene fermata il campo RATE non viene usato, ma, naturalmente, il contatore ROW viene incrementato di due per ogni ciclo eseguito.

Elaborazione Distribuita

La capacità, in una *rete* di CPU, di condividere delle risorse è una caratteristica avanzata della filosofia dello Z8000. In parole povere questa possibilità viene definita *elaborazione distribuita.*

L'elaborazione distribuita è una modalità che permette di gestire, per mezzo di piccoli programmi, delle grosse applicazioni. Il lavoro viene suddiviso in *moduli*, e ciascuno di essi ha il proprio piccolo sistema. Quando è necessario i piccoli sistemi parlano l'un con l'altro, ma per la maggior parte del tempo eseguono il proprio lavoro in modo indipendente.

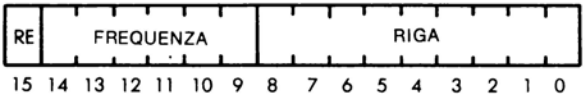


Fig. 2.10 — Localizzazione dei Campi del Registro REFRESH

Molti piccoli sistemi sono meno complicati di uno grosso? Qui di seguito riportiamo un «argomento plausibile». Si consideri un sistema che porti avanti 20 cose diverse. La matematica elementare dice che ci sono

$$\frac{20 \times 19}{2} = 190$$

possibili interazioni tra coppie di questi (dimentichiamoci delle interazioni triple, ecc.). Adesso supponete che queste 20 cose siano separate in cin-

que gruppi di quattro cose in modo che esistano interazioni all'interno di ciascun gruppo e tra i gruppi interi, ma che non si verifichi nulla tra le parti di un gruppo e quelle di un altro. Ci sono quindi

$$\frac{4 \times 3}{2} = 6$$

interazioni all'interno di ciascun gruppo e

$$\frac{5 \times 4}{2} = 10$$

interazioni tra i diversi gruppi per un totale di

$$5 \times 6 + 10 = 40$$

possibili interazioni.

Per cui ora invece di 190 interazioni ne abbiamo solo 40 di cui preoccuparci. Suddividendo il problema in cinque moduli la complessità si è ridotta di circa un fattore cinque. (Esiste una legge generale in proposito?).

L'architettura dello Z8000 è pensata in modo tale che quando i 20 lavori sono stati divisi in cinque gruppi di quattro, ciascun gruppo dovrebbe avere una propria CPU, un po' di memoria, una interfaccia I/O e così via. Si supponga per esempio che questi gruppi debbano condividere le risorse di una stampante di linea.

Siccome i vari processi probabilmente richiedono che le uscite, verso stampante non siano mescolate, allora lo stato «busy» (occupato) della stampante di linea non è sufficiente per controllarne l'accesso. Per cui, ogni CPU ha bisogno di un qualche mezzo che permetta il controllo della stampante di linea, ma che impedisca alle altre CPU di usarla, anche se la stampante sembra in uno stato di attesa.

Il meccanismo che permette di implementare questa tecnica usa l'uscita MO dello Z8000, l'ingresso MI e un bus esterno a 4-bit con una logica esterna che faccia corrispondere le quattro linee del bus ai due piedini della CPU. Non ci addentriamo nel protocollo del bus e nella logica esterna. Dovremo semplicemente stabilire il protocollo interno che le CPU originano:

(1) Se si vuole utilizzare la risorsa condivisa, si guardi alla propria linea MI. Se si vede un segnale di «in-uso», andatevene e riprovate più tardi. Altrimenti posizionate la vostra linea MO.
(2) Si aspetti che il proprio segnale si propaghi nel bus; poi si guardi nuovamente MI.
(3) Se si riscontra un segnale «in-uso», questo significa che il nostro segnale si è propagato attraverso la *daisy-chain* (catena di priorità).

Si procede all'utilizzo della risorsa. Altrimenti si azzeri la propria linea MO e si riprovi più tardi.

(4) Dopo aver finito di utilizzare la risorsa, si azzeri la propria linea MO.

Il Dispositivo CPU

I 48 segnali disponibili ai piedini dello Z8001 (o 40 per lo Z8002) possono essere suddivisi in vari gruppi, che sono stati discussi o ai quali si è fatto riferimento in questo capitolo. (Vedasi Figura 2.11).

- *Bus Indirizzi/Dati*: Le linee AD_{15} – AD_0 sono condivise come linea dati e indirizzi; entrambe sono usate per accessi alle memorie o ad I/O. (Vedasi Capitoli VI e VII).
- *Bus Controllo*: Esistono tre uscite della CPU che ci dicono che cosa c'è sul bus e che segnalano quando questo è valido.

  $\overline{AS}$: strobe indirizzo – indica che gli indirizzi sul bus sono validi.

  $\overline{MREQ}$: richiesta memoria – indica che le linee del bus contengono un indirizzo di memoria, altrimenti è un indirizzo di I/O.

  $\overline{DS}$: strobe dato – per la temporizzazione dei dati in I/O.

  Esistono anche due linee di *handshaking* (sincronizzazione di eventi fra due dispositivi) che permettono ad altri dispositivi, oltre alla CPU, di usare il bus.

  $\overline{BUSRQ}$: richiesta del bus – inviato alla CPU per richiedere l'utilizzo del bus.

  $\overline{BUSAK}$: riconoscimento del bus – inviato dalla CPU per garantire il controllo al richiedente.
- *Controllo CPU*: ci sono due linee di ingresso che permettono il controllo della CPU da parte di dispositivi esterni.

  $\overline{WAIT}$: attesa – indica che il dispositivo di I/O o l'unità di memoria non è ancora pronta per il trasferimento del dato.

  $\overline{STOP}$: stop – pone la CPU in uno stato di stop (utile per realizzare il modo di debug *passo-passo*).
- *Stato*: Esistono sette segnali di stato inviati in uscita dalla CPU.

  $R/\overline{W}$: lettura/scrittura – informa se la CPU sta leggendo o sta scrivendo in memoria.

  $B/\overline{W}$: byte/parola – informa se sul bus si trova una richiesta espressa in un byte o in una parola.

N/S: normale/sistema — informa se la CPU è nel modo normale o sistema.

ST_3–ST_0: stati — codifica 12 stati della CPU (Vedasi Figura 2.12).

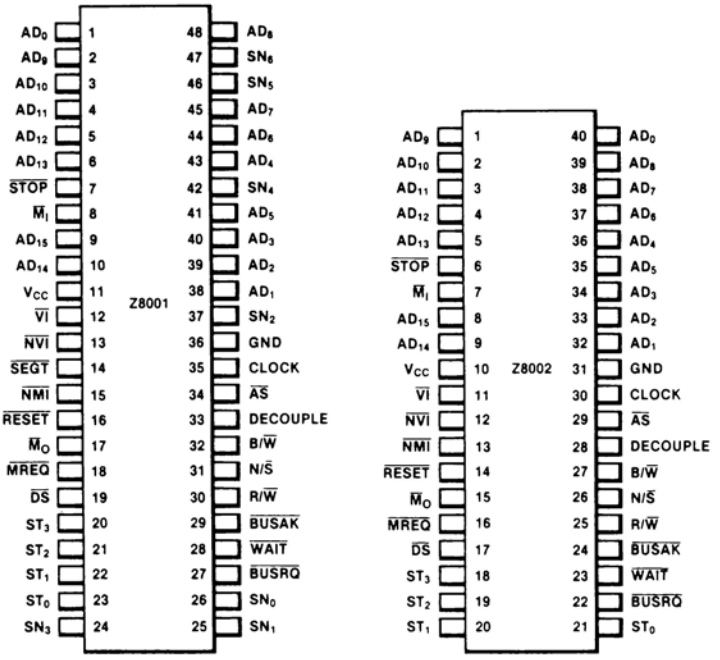


Figura 2.11 — Denominazione dei Piedini dello Z8001 e Z8002 (Per Cortesia della Zilog, Inc.)

Significato di ST_3–ST_0 (esadec.)	Significato di ST_3–ST_0 (esadec.)
0 operazione interna	8 area memoria dati
1 ciclo di rinfresco	9 area memoria di stack
2 richiesta di I/O	A (inutilizzato)
3 richiesta di I/O speciale	B (inutilizzato)
4 riconoscimento: trappola da segmento	C area di memoria programma (prima parola dell'istruzione)
5 riconoscimento: NMI	D area memoria programma (parole successive)
6 riconoscimento: NVI	E (inutilizzato)
7 riconoscimento: VI	F (inutilizzato)

Figura 2.12 — Codifica delle Uscite di Stato dello Z8000

- *Interruzioni*: Esiste una linea per ciascun tipo di interruzione.
  - $\overline{NMI}$: interruzione non mascherabile
  - $\overline{NVI}$: interruzione non vettoriale
  - $\overline{VI}$: interruzione vettoriale
- *Sincronizzazione dell'elaborazione distribuita*: sono le linee MI e MO discusse prima.
  - $\overline{MI}$: ingresso multi-micro – indica che qualche microprocessore richiede la risorsa condivisa.
  - $\overline{MO}$: uscita multi-micro – indica che questo processore richiede la risorsa condivisa.
- *Supporto esterno dei dispositivo*
  - V_{cc}: +5 Volt
  - GND = massa
  - CLK = clock – ingresso di una singola fase a 5 Volt
- *Inizializzazione*:
  - $\overline{RESET}$: azzeramento – forza la CPU in uno stato iniziale; dalla locazione 2 del formato dello stack del segmento zero viene preso lo stato della CPU.
- *Segmentazione della memoria* (solamente nello Z8001):
  - SN_6–SN_0: numero del segmento – in uscita da CPU verso MMU.
  - $\overline{SEGT}$: trappola dovuta al segmento – segnale inviato da MMU a CPU per indicare che la richiesta è inconsistente con la tabella interna della MMU.

Capitolo III

Introduzione alle Tecniche di Programmazione

Questo capitolo costituisce l'introduzione per la scrittura di effettivi programmi per lo Z8000. Come già detto nel Capitolo II, dovrete sforzarvi di comprendere quanto viene illustrato rileggendo, se necessario, le parti meno chiare, ma se dopo questo tentativo vi rimane ancora qualche cosa di oscuro proseguite normalmente. Non è semplice presentare questo soggetto in modo lineare, per cui alcuni argomenti vi saranno più chiari dopo che avrete letto i prossimi capitoli.

Siccome la trattazione non si riferisce ad un particolare assemblatore dello Z8000, non ci sono costruzioni del tipo

do.....od

if.....then.....else

ed altre che sono in genere disponibili. Nel Capitolo X discuteremo gli strumenti di sviluppo esistenti.

Inizieremo questo capitolo con un algoritmo modesto ma non banale di cui seguiremo dall'inizio alla fine la sequenza dei passi necessari per produrre un programma completo che lo implementi. L'algoritmo serve per crittare un blocco di un testo in modo tale che sia incomprensibile a qualsiasi persona che cerchi di controllarlo. L'idea dell'algoritmo proviene dal programma di «crittografia» RATFOR presentato nel libro, molto utile, *Software Tools* di Kerninghan e Plauger.

La crittografia è semplicemente il processo che trasforma un messaggio riconoscibile in uno non riconoscibile in modo tale che solo il ricevente designato sia in grado di decodificarlo nel modo corretto, e che nessun

altro, al quale accada di vederlo, sia in grado di comprenderne il significato. Per esempio, considerate il messaggio «meet me at eight — vediamoci alle otto». Supponiamo di decidere di rimpiazzare ciascuna lettera del messaggio con quella successiva dell'alfabeto, per cui m viene sostituito con n, e con f, t con u, ecc. Il messaggio diviene «nffu nf bu fjhiu». Il destinatario che conosce lo schema crittografico usato può «decodificare» il messaggio lavorando all'indietro — sostituendo ciascuna lettera con quella precedente dell'alfabeto.

La regola crittografica scritta sopra è così semplice che un esperto di codici potrebbe decifrare il messaggio in pochi minuti senza che gli sia stata detta la regola in anticipo. Uno dei motivi della semplicità è dovuto al fatto che la stessa regola viene applicata ad ogni lettera. Nel momento in cui scoprite che f è la codifica di e, avete trovato quattro delle tredici lettere. Sapete a questo punto che il messaggio ha la seguente forma

«_ee_ _e __ e____»

Poi, pensando alle possibili parole di due lettere che terminano con «e» e notando che le prime due parole iniziano con la stessa lettera, arriverete rapidamente a

«meet me _t e___t»

E come si è trovata la «e» da cui partire, ora dovete solamente provare le varie lettere che possono apparire come doppie nel mezzo di una parola di quattro lettere.

La regola crittografica del nostro algoritmo è un poco più difficile di quella descritta sopra; infatti nel nostro caso si applicano diverse regole di crittografia. In questo caso si applica la prima regola alla prima lettera del messaggio, la seconda alla seconda lettera, la terza alla terza lettera; poi si ritorna ad usare lo stesso procedimento applicando la prima regola alla quarta lettera, la seconda alla quinta lettera e così via.

Il ricevente deve comprendere le diverse regole applicate alle diverse posizioni, altrimenti non avrà vita facile nel decifrare il messaggio. Per cui, si fa corrispondere ad ogni regola un nome costituito da una lettera: regola—A, regola—B ecc. Quindi il modo più rapido per comunicare al ricevente quale sia la regola utilizzata, consiste nell'inviare l'insieme delle lettere d'identificazione riunite in un'unica parola chiamata *chiave*. Per esempio se la chiave è «CAT» significa che la regola—C è usata sulla prima, quarta, settima, ecc. posizione; la regola—A è usata sulla seconda, quinta, ottava, ecc.; la regola—T è usata sulla terza, sesta, nona, ecc.

L'ultima cosa da fare, priva di ogni complessità, è definire la regola–A, regola–B, ecc. In questo modo il ricevente, invece di dover consultare il libro che spiega la regola–A, regola–B ecc., ha solo bisogno di conoscere la formula utilizzata per derivare la regola–A dal codice ASCII di A, e così via. La formula reale usata nel nostro algoritmo, è basata su di un semplice calcolo chiamato *OR-esclusivo*, che descriveremo nell'illustrazione dell'algoritmo.

Nella Figura 3.1 è illustrato l'algoritmo, identificato come Algoritmo E, che illustreremo punto per punto.

Passo 1. La chiave è un insieme di lettere (esempio «abracadabra») che costituisce la parte segreta della regola di crittografia. Ogni conoscitore della chiave può decifrare il testo crittato.

Passo 2. La chiave ed il blocco di testo vengono memorizzati in codice ASCII. Ciascuno di essi occupa un insieme di byte di memoria adiacenti ed è terminato con un byte zero. (Lo zero è un buon terminatore perché corrisponde al carattere ASCII nul-insignificante, che non farà mai parte di una effettiva stringa di testo). Il terminatore viene utilizzato per segnalare il completamento dell'elaborazione di una stringa. In alternativa al terminatore può essere fornito, per ciascuna stringa, il numero di caratteri che la compone. Questo metodo è più difficile da usare ed è inoltre più soggetto ad errori di quanto non lo sia il metodo visto sopra.

Passo 3. Un indice ed un puntatore sono esempi di due utilizzi dei regi-

Algoritmo per la Crittografia di un Testo

1. Scegliete una chiave ed una stringa del testo. Supponiamo che N sia il numero di caratteri contenuto nella stringa.
2. Supponete di avere un testo da crittografare. Supponete che il testo sia memorizzato in byte adiacenti di memoria in codice ASCII con un byte zero che indichi la fine del testo.
3. Inizializzate l'indice I ad 1: inizializzate il puntatore P in modo da puntare al primo carattere da crittografare.
4. Prelevate il carattere puntato da P.
5. Se il carattere è uno zero fermatevi perché avete terminato la crittografia. Altrimenti eseguite l'OR esclusivo del carattere prelevato con l'I-esimo carattere della chiave.
6. Memorizzate, per mezzo di P, il carattere crittografato.
7. Incrementate P dell'indirizzo di un byte; sostituite I con (I mod N) + 1.
8. Ritornate al punto 4.

Figura 3.1 — Algoritmo E

stri dello Z8000 discussi nel Capitolo II: un registro indice ed un registro indirizzo.

Passo 4. Ottenere il carattere indicato dal puntatore P significa solamente che si sta utilizzando un registro indirizzo per specificare il dato desiderato.

Passo 5. Un'operazione fondamentale nello Z8000 è il controllo che permette di verificare se un carattere è zero. Questo ed altri controlli condizionali sono la base di tutta la complessità e potenza dei programmi dei calcolatori.

L'OR esclusivo è un'operazione logica analoga alla sottrazione. In effetti qualche volta è anche chiamato *differenza simmetrica*. L'OR esclusivo (XOR), per due operandi ciascuno di un bit è definito nel modo seguente:

$$(0 \text{ XOR } 0) = (1 \text{ XOR } 1) = 0$$

$$(1 \text{ XOR } 0) = (0 \text{ XOR } 1) = 1$$

Per operandi più lunghi, la stessa definizione si applica nei due operandi ai bit di posizione corrispondenti. Per indicare l'operazione di OR esclusivo usiamo il simbolo ∨. Perciò, per esempio, F∨3 = C; 3∨5 = 6.

ESERCIZIO 1: Quanto fanno 48∨0; 48∨48; F0∨0F?

Per ogni argomento A, B, C l'operazione di OR esclusivo ubbidisce alle seguenti regole:

(1) A∨A = 0
(2) A∨0 = A
(3) (A∨B)∨C = A∨(B∨C)
(4) Se B∨A = C∨A, allora B = C
(5) A∨B = B∨A

Pur essendo interessanti, non ci interessa comprendere le proprietà dell'OR esclusivo, non essendo esse essenziali per la comprensione dell'algoritmo e dei successivi programmi. Però, se siete portati alla matematica, potete notare che la regola (4) è quella che permette di realizzare la crittografia con l'OR esclusivo ∨: cioé lettere diverse non sono mai codificate nella stessa lettera crittografica. Inoltre, le regole (1), (2) e (3) mostrano che essendo (X∨A)∨A = X∨(A∨A) = X∨0 = X, per «decifrare» il nostro testo occorre semplicemente crittografarlo nuovamente.

Passo 6. Questo è l'immagine speculare del passo 4. Il carattere indirizzato dal puntatore P viene sostituito da quello crittografato; nel Passo 4

era stato prelevato e posto nella CPU per l'elaborazione.

Passo 7. Spieghiamo infine la funzione *mod* (I mod N) è definito come il numero compreso tra 0 e N − 1 corrispondente al resto che si ottiene quando I è diviso per N. Per esempio, (13 mod 12) = 1; (6 mod 6) = 0; (−9 mod 4) = 3.

ESERCIZIO 2: Quanto fa (4 mod 3)?

Il passo «sostituite I con (I mod N) + 1» fa assumere ad I, in modo ciclico, i valori da 1 a N; 1, 2, ..., N, 1, 2, ..., N, ..., proprio come le cifre orarie di un orologio digitale che si ripetono ciclicamente da 1, ..., 12, 1, ..., 12, ...

Diamo ora un esempio. Supponiamo che il nostro testo sia costituito dal codice ASCII dei caratteri della frase seguente:

«Hi! I'm a Z8000.»

e la nostra chiave sia:

«abracadabra»

Il testo è costituito dai codici: 48, 69, 21, 20, 49, 27, 6D, 20, 61, 20, 5A, 38, 30, 30, 30, 2E, 0. La chiave è costituita da: 61, 62, 72, 61, 63, 61, 64, 61, 62, 72, 61, 0. (Vedasi Figura 1.15). Dopo aver applicato il nostro algoritmo, il testo crittato è costituito da: 48¥61, 69¥62, 21¥72, 20¥61, 49¥63, 27¥61, 6D¥64, 20¥61, 61¥62, 20¥72, 5A¥61, 38¥61, 30¥62, 30¥72, 30¥61, 2E¥63,0.

ESERCIZIO 3: Scrivete la stringa crittografata (esempio eseguire le operazioni ¥). Come appare il testo crittografato?

Nella Figura 2 è riportato un programma per lo Z8000 (non-segmentato) che implementa l'algoritmo E. Per prima cosa si noti che sta in una pagina. Questa è una regola empirica che è diventata recentemente popolare — un programma che non può stare in una pagina è troppo grande e dovrebbe essere diviso in parti più piccole.

Si noti poi il notevole blocco di testo all'inizio racchiuso tra i due «!». Questi blocchi sono chiamati *commenti*; essi sono usati per comodità del lettore — il programma di assemblaggio (assembler) non considera i commenti quando il programma viene convertito in codice macchina.

I commenti ci forniscono informazioni essenziali per ricollegare il programma all'Algoritmo E. Per prima cosa, abbiamo detto dove sono memorizzati nella memoria la chiave e il testo da crittografare. Poi abbiamo detto quali sono i registri dello Z8000 che devono corrispondere ai simboli P, I ed N dell'algoritmo. Infine, ciascun passo dell'algoritmo viene identificato con un piccolo gruppo di istruzioni.

!Programma per Crittografare con l'algoritmo E.

Supponete che sia stata scelta la chiave e che sia memorizzata partendo dall'indirizzo KEY, con un byte zero che segnala la fine.

Supponete che il testo da crittografare sia memorizzato partendo dalla locazione SECRET.

Useremo le seguenti assegnazioni dei registri:

R0: registro di lavoro
R1: puntatore P dell'algoritmo
R2: indice I dell'algoritmo (considerato come I−1)
R3: utilizzato per memorizzare la lunghezza della chiave N!

```
        LDA R1,KEY          !Calcola la lunghezza della chiave!
        CALL LENGTH
        LD R3,R0            !Salva la lunghezza (N)!

        LDK R2;#0           !Passo 3 dell'Algoritmo E!
        LDA R1,SECRET

LOOP:   LDB RL0,@R1         !Passo 4!

        TESTB RL0           !Passo 5!
          JR Z,DONE
        XORB RL0,KEY(R2)

        LDB @R1,RL0         !Passo 6!

        INC R1              !Passo 7!
        INC R2
        CALL MOD23

        JR LOOP             !Passo 8!

DONE:   HALT                !Ancora una parte del Passo 5!
```

Figura 3.2 — Programma dell'Algoritmo E

Nel Capitolo II si è parlato dei registri, ma non abbiamo ancora trattato gli indirizzi simbolici delle locazioni di memoria. Fondamentalmente, ciascun byte della memoria dello Z8000 ha un indirizzo: un numero compreso tra 0 e 65.535 (se lo Z8000 lavora in modo segmentato c'è un numero addizionale, relativo al segmento, compreso tra 0 e 127). Supponiamo che la chiave sia «CAT» e che sia memorizzata nei byte della memoria di locazioni 18, 19, 1A, 1B (3 caratteri più un terminatore). Quindi la locazione 18 contiene un 43 (il codice ASCII di C), 19 contiene 41, 1A contiene 54 e 1B contiene zero (Vedasi Figura 1.15). Inoltre, supponiamo che

il testo da crittografare sia memorizzato partendo dal byte di memoria di locazione 20F6 fino a 2117. Se i primissimi caratteri sono «Now is the time ...» allora 20F6 contiene 4E, 20F7 contiene 6F, 20F8 contiene 77 20F9 contiene 20, e così via.

Durante l'esecuzione del programma, lo Z8000 avrà bisogno di caricare «18» in un registro indirizzo quando dovrà lavorare con la chiave e «20F6» quando dovrà lavorare con il testo da crittografare. Per far sì che il programmatore non lavori con gli indirizzi reali della memoria, e in generale per evitare la «gestione» a livello numerico delle allocazioni della memoria disponibili (esempio: 18 e 20F6), si sono escogitate varie tecniche. La tecnica più avanzata è *l'indirizzamento simbolico.* I commenti ci dicono che la chiave è memorizzata partendo dall'indirizzo KEY e che il testo da crittografare è memorizzato partendo dalla locazione SECRET. Questo significa che da qualche parte al simbolo KEY è stato fatto corrispondere l'indirizzo 18 e al simbolo SECRET l'indirizzo 20F6. Si dice che KEY ha il valore 18; SECRET ha il valore 2016.

Quanto sopra descritto può essere realizzato in diversi modi. Per esempio, il nostro programma potrebbe contenere le istruzioni

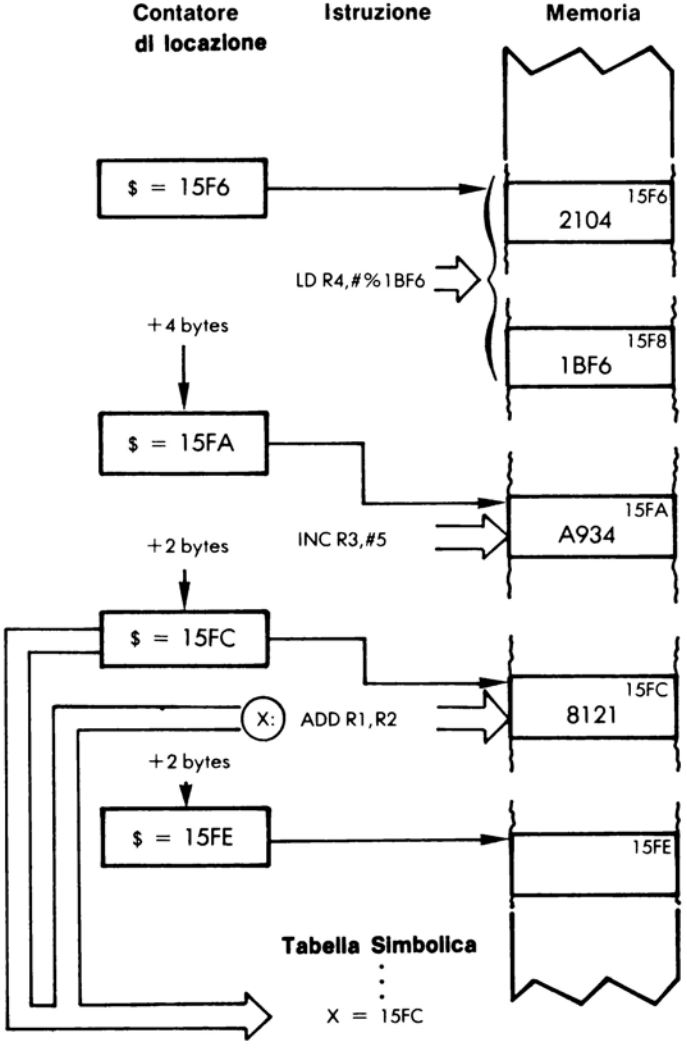
```
KEY = %18

SECRET = %20F6
```

(Il % viene utilizzato per far sapere all'assemblatore che 18 è esadecimale e non decimale). Questo è il modo più semplice, ma costringe il programmatore a scegliere le locazioni. Dovremmo trovare un metodo automatico per quando proseguiremo. Il metodo base richiede l'uso del *contatore di locazione.*

Quando il programma di assemblaggio traduce il nostro programma sorgente deve ricordarsi dove viene posta in memoria ogni istruzione assemblata; per fare questo utilizza un contatore di locazione. Nell'istante in cui parte, il contatore di locazione contiene l'indirizzo della prima locazione di memoria che può essere usata dal programma (più avanti diremo come esso trovi dove sarà la prima locazione). Poi, quando un'istruzione è assemblata, il numero di byte che occupa è sommato al contatore di locazioni per ottenere l'indirizzo relativo alla prossima istruzione. La Figura 3.3 illustra come l'assemblatore utilizza il contatore di locazione. Nei Capitoli IV e V sarà descritto come l'assemblatore decide il numero di byte corrispondenti ad ogni istruzione e quali sono i reali valori esadecimali di queste. La linea contenente

```
X: ADD R1,R2
```

illustra il modo con cui i nomi simbolici vengono assegnati alle locazioni di memoria. Quando il programmatore scrive queste linee, invia un comando all'assemblatore che assegna ad X un valore uguale all'indirizzo di memoria in cui viene memorizzata la codifica esadecimale di ADD R1,R2. Dal momento che il codice esadecimale di ADD R1,R2



Il valore assegnato a X è il valore corrente del contatore di locazione

Figura 3.3 — Uso del Contatore di Locazione in Fase di Assemblaggio

(8121) è memorizzato alla locazione 15FC (Vedasi Figura 3.3), ad X viene assegnato il valore 15FC.

Naturalmente, viene spontanea una domanda: «Perché abbiamo nomi simbolici per le locazioni di memoria ma non per i registri?». La risposta è che ai registri si possono assegnare nomi simbolici (con la maggior parte degli assemblatori) con istruzioni aventi la seguente forma:

P = R1

N = R3

Negli ultimi capitoli faremo la stessa cosa, ma nei nostri programmi crea minor confusione l'utilizzo esplicito dei nomi dei registri.

Nel seguito sarà chiarito, per mezzo di esempi, l'uso dei simboli nei programmi dei calcolatori. Per ora, ritorniamo ad esaminare il programma di Figura 3.2.

Il programma è costituito da 15 istruzioni. Le prime cinque e l'ultima sono eseguite una sola volta ciascuna. Le altre nove costituiscono l'anello principale.

Tre di queste (LDB RL0,@R1; XORB RL0,KEY(R2); LDB@R1, RL0) esprimono quello che fa l'anello. Le rimanenti sei riguardano il meccanismo di funzionamento e la chiusura dell'anello.

Gli anelli (loop) possiedono strutture diverse. In quello di Figura 3.2 all'inizio c'è il test che chiude il loop (TESTB RL0; JR Z,DONE), mentre alla fine c'è l'aggiornamento dei parametri del loop (P ed I per l'Algoritmo E). Si notino le due interessanti conseguenze che ne derivano: (1) è possibile terminare il loop senza averlo mai eseguito; (2) i parametri del loop (P ed I) hanno sempre dei valori «naturali»; per esempio, i valori che definiscono l'ennesimo giro del loop, ma quando il loop termina i valori da essi espressi corrispondono a quelli di un giro mai avvenuto.

Descriviamo ora le istruzioni. Le prime tre determinano la lunghezza della chiave - il parametro N dell'Algoritmo E. L'istruzione

LDA R1,KEY

carica l'indirizzo, in cui si trova la chiave, nel registro 1. Si noti come questa sia diversa dall'istruzione LD R1,KEY che caricherebbe il *contenuto* della locazione di memoria, il cui nome simbolico è KEY, in R1; per esempio, i primi due caratteri della chiave. Quindi se KEY ha il valore dell'esempio sopra riportato, viene caricato in R1 il valore esadecimale 18.

L'istruzione

CALL LENGTH

è la chiamata ad una *subroutine* memorizzata partendo dalla locazione il cui nome simbolico è LENGTH. Per ora questa subroutine non esiste; la scriveremo dopo. Per adesso sappiamo che ci occorre la lunghezza della chiave da usare nel nostro algoritmo, per cui scriviamo solo la chiamata di questa subroutine e procediamo. Dal momento che la routine LENGTH oltre ad essere facile da scrivere, è anche utile per altre occasioni, l'esperienza ci consiglia di richiedere che questa riceva in R1 l'indirizzo di una stringa terminata con zero e riporti in R0 la sua lunghezza (escluso lo zero). Adesso, mentre stiamo ancora lavorando sulla struttura generale del programma, non abbiamo bisogno di pensare ai dettagli di LENGTH.

L'istruzione

```
LD R3,R0
```

muove soltanto la lunghezza della chiave da R0 a R3. Riducendo l'utilità generale di LENGTH avremmo potuto eliminare questa istruzione facendo porre in R3 la risposta della subroutine LENGTH.

Le due istruzioni successive costituiscono il Passo 3 dell'Algoritmo E. L'istruzione:

```
LDK R2,#0
```

carica il valore 0 in R2 (ricordate: R2 contiene I − 1). Il simbolo «#» identifica un argomento *immediato*. Questo significa che il valore che deve essere caricato in R0 (esempio 0) fa parte dell'istruzione. LDK R2,#0 è un'alternativa di LD R2,#0; quest'ultima istruzione fa esattamente la stessa cosa ma occupa una quantità doppia di memoria. LDK può essere usata solamente con valori compresi tra #0 e #15.

Casualmente, quando scriviamo in linguaggio di assemblaggio dei numeri nei nostri programmi, questi sono numeri decimali. Quando scriviamo dei numeri esadecimali, questi sono preceduti dal segno «%». Per esempio, 15 = %F.

L'istruzione

```
LDA R1,SECRET
```

è analoga all'istruzione LDA R1,KEY discussa prima. Essa inizializza P per puntare al primo carattere del testo da crittografare. Usando il valore dell'esempio sopra descritto, viene caricato in R1 il valore %20F6.

L'istruzione

```
LOOP: LDB RL0,@R1
```

introduce molte nuove caratteristiche del linguaggio di assemblaggio. Per prima cosa, il nome «LOOP» seguito da «:» indica che la locazione di memoria contenente l'istruzione LDB RL0,@R1 sarà contrassegnata con il nome simbolico LOOP. (Vedasi Figura 3.3). In questo modo, quando più avanti nel programma desidereremo ripetere questo passo, potremo usare l'istruzione

JR LOOP

per tornare indietro. La «@» in LDB RL0,@R1 indica che R1 deve essere usato come un registro indirizzo; contiene l'indirizzo di memoria del byte che deve essere caricato nel registro a byte RL0. RL0 è la metà meno significativa di R0. La «B» alla fine di LDB indica un'istruzione che opera sul byte. Tutte le istruzioni che operano sul byte finiscono in «B», eccetto DBJNZ, e tutte le istruzioni che operano sulle parola-lunghe finiscono in «L». L'opposto non è vero. Per esempio, SUB-sottrazione — non è un'istruzione sul byte, e SLL-shift left logical — scorrimento logico a sinistra — non è un'istruzione con riferimento ad una parola lunga.

Le tre istruzioni successive e l'ultima istruzione costituiscono il Passo 5 dell'Algoritmo E. Il passo 5 ha la forma

IF (*così e così*) THEN DO (*una cosa*) ELSE DO (*l'altra*)

Questa è una forma molto importante nella programmazione, perché può essere scritta in un passo, eliminando attorno al nostro algoritmo molti meccanismi di salto. Dal momento che possiamo vedere in una sola volta quando fare «una cosa» e quando fare «l'altra», si semplifica, rendendolo più facile da leggere, il *flusso di controllo.*

Sfortunatamente, la maggior parte dei calcolatori non possiede istruzioni di questo tipo, quindi dobbiamo implementarle utilizzando «salti meccanici». Le istruzioni

```
        TESTB RL0
          JR Z,DONE
            •
            •
            •
DONE  :  HALT
```

implementano la parte del Passo 5 «se il carattere è un byte zero, fermati». L'istruzione JR Z,DONE significa: se l'operazione precedente ha fatto si che il bit Z sia posto ad uno (indicando un risultato nullo) allora inizia ad

eseguire l'istruzione memorizzata alla locazione di memoria il cui nome simbolico è «DONE». L'«operazione precedente» a cui siamo interessati è LDB RL0, @ R1, questa carica RL0 con un byte che potrebbe essere zero. Ma, diversamente dall'istruzione di spostamento del PDP-11, *l'istruzione di caricamento non cambia nessun bit di condizione*, per cui LDAB RL0, @ R1 deve essere seguita da una TESTB RL0 prima di eseguire un controllo per vedere se c'è un byte a zero.

La parte «ELSE» del Passo 5 è realizzata dall'istruzione

```
XORB RL0,KEY(R2)
```

In questa istruzione si utilizza il modo di indirizzamento *indicizzato*, che discuteremo nel Capitolo V; l'operando «KEY(R2)» indica il byte (essendo XORB un'istruzione riferita al byte) il cui indirizzo di memoria è determinato sommando il contenuto di R2 all'indirizzo il cui nome simbolico è KEY. Perciò, se R2 = 0, otteniamo il primo carattere della chiave; se R2 = 1 otteniamo il secondo, e così via. In altre parole, per corrispondere all'algoritmo, R2 deve contenere I-1. Tutta l'istruzione permette di realizzare l'OR esclusivo del contenuto di RL0 (il carattere del testo appena estratto) e il carattere I-esimo della chiave. C'é un altro modo per disporre le istruzioni del Passo 5 in modo da averle tutte insieme:

```
            TESTB RL0
            JR NZ,NOTDONE
            HALT
NOTDONE:    XORB RL0,KEY(R2)
```

È solo una questione di preferenza. La forma originale è migliore per diverse ragioni:

- La condizione NZ è l'opposto di quella fissata dall'algoritmo; è difficile comprendere le doppie negazioni.
- L'etichetta (label) NOTDONE è totalmente artificiale. Non esiste alcuna ragione per avere una label nel mezzo di questo loop.
- È più chiaro far si che la terminazione del programma si verifichi alla fine.
- Successive modificazioni del programma potrebbero introdurre altre condizioni di terminazione. L'istruzione con label finale, HALT, garantisce un modo naturale per esprimere queste terminazioni.

L'istruzione successiva

```
LDB @R1,RL0
```

implementa il Passo 6. È esattamente l'opposto di LDB RL0,@R1 utilizzata per prelevare il byte. Questa istruzione permette di memorizzare il byte crittografato sul byte originale appena crittografato, distruggendolo. Questo fatto evidenzia un'altra piccola sorprendente caratteristica della denominazione delle istruzioni dello Z8000.

Nei primi calcolatori il significato originale del *caricamento* era il seguente: trasferire il contenuto di una specifica locazione in uno specifico (o immediato) registro. L'operazione corrispondente, sempre presente, detta *memorizzazione* (store), operava in modo opposto. In questi calcolatori c'erano istruzioni separate per il trasferimento da un registro ad un altro o (qualche volta) da una locazione di memoria ad un'altra. Queste istruzioni permettevano di spostare dei dati da un luogo ad un altro, e in alcune delle ultime macchine (esempio il PDP-11) queste operazioni erano tutte combinate in una unica istruzione — la istruzione di «*move*» (spostamento).

L'istruzione di caricamento dello Z8000 è in verità molto più simile ad una istruzione di move che ad una istruzione tradizionale di caricamento. I programmatori esperti dovranno disimparare qualche cosa per riuscire a pensare in termini di caricamento della memoria da un registro.

Le due istruzioni successive, implementano il Passo 7 dell'Algoritmo E. La prima,

```
INC R1
```

fa sì che il puntatore P indichi il prossimo byte del testo da crittografare. Questa istruzione è un esempio di una condizione di *default* (la condizione di default di una variabile è quella che viene assunta automaticamente quando non viene fornito un valore). In verità la forma completa dell'istruzione indica una quantità di incremento compresa tra 1 e 16. Per esempio, se piuttosto che ai byte stava puntando alle parole, avremmo potuto scrivere INC R1,#2, per far sì che il nostro puntatore puntasse alla prossima parola. Ma se non esprimiamo il valore dell'incremento, come indicato sopra, l'assemblatore assume un valore di default: INC R1,#1. Questa è la dimostrazione di una buona *ingegnerizzazione.*

Le due istruzioni seguenti:

```
INC R2
CALL MOD23
```

corrispondono a «sostituisci I con (I mod N) + 1». Ricordatevi che, nell'algoritmo, l'indice I è compreso tra 1 e N, mentre in questo programma

vogliamo che sia compreso tra 0 ed N—1. Perciò, invece di partire da 1, partiamo da zero, ed invece di (I mod N) + 1 calcoliamo (I + 1) mod N. (Siete capaci di capire perchè?).

L'istruzione INC R2 calcola I + 1, mentre CALL MOD23 ci da (I + 1) mod N. Questo si verifica perchè la subroutine MOD23, non ancora scritta, calcola

(contenuto di R2) *mod* (contenuto di R3)

e lascia il risultato in R2.

Probabilmente MOD23 non è una subroutine molto usata. Potreste desiderare che la subroutine mod funzionasse in molti modi diversi — in funzione di dove essa trova i suoi argomenti e di dove lascia le sue risposte — perciò potremmo scrivere anche una routine speciale che utilizzi i registri in cui i nostri programmi hanno gli argomenti. MOD23 probabilmente non sarà mai chiamata da nessuna parte eccetto che da questo punto del programma, ma invece di codificarla linea per linea l'abbiamo realizzata come subroutine per quattro ragioni:

- La struttura del programma riflette meglio l'algoritmo se questi dettagli sono posposti ad un livello più basso ed inoltre il programma è più facile da leggere e capire;
- Quando si sta ancora cercando di strutturare il problema principale si è distratti dall'analisi dei dettagli di un sottoproblema;
- Definire come il programma principale colloquierà con questa subroutine aiuta a chiarire la struttura logica sia a livello subroutine che a livello di programma principale ed elimina le possibli confusioni che si originano considerando distintamente questi problemi;
- Questa organizzazione serve a semplificare e chiarire il flusso di controllo — affrontando perciò la maggior causa di problemi nel progetto e manutenzione di programmi.

L'istruzione successiva,

JR LOOP

ci riporta al Passo 4 (esattamente come dice di fare il Passo 8). Questa è una istruzione dello stesso tipo di JR Z,DONE usata prima; quando non è specificata la condizione (esempio Z), l'assemblatore genera la forma dell'istruzione che permette sempre di saltare alla locazione specificata.

L'istruzione JR ha una importante limitazione. L'indirizzo a cui si vuole saltare non deve essere più di 254 byte prima o 256 byte dopo l'indirizzo della locazione contenente il JR. Quanto detto sopra non origina problemi se scrivete un programma che, secondo la regola empirica, stia in una pagina — siccome non esiste, in generale, una ragione per saltare (in

opposizione alle CALL) in un punto qualsiasi al di fuori del vostro programma. (Costituisce una eccezione la locazione a cui si salta quando si trovano errori «fatali»).

Se dovete eseguire un salto «lungo», esiste una forma speciale dell'istruzione JR chiamata JP, che vi permette di saltare in un qualsiasi punto della memoria, però JP occupa quattro (o anche sei) byte di memoria invece dei due byte utilizzati da JR. JP viene anche utilizzata con le tecniche di programmazione avanzate (vedasi Figura 6.13), siccome permette salti ad indirizzi specificati in un registro indirizzo o ad una tabella di istruzioni indicizzata da un registro indice.

Con questo termina il nostro programma – tranne le due routine che abbiamo lasciato da parte per scriverle in un tempo successivo: LENGTH e MOD23. Adesso affrontiamo questi problemi minori.

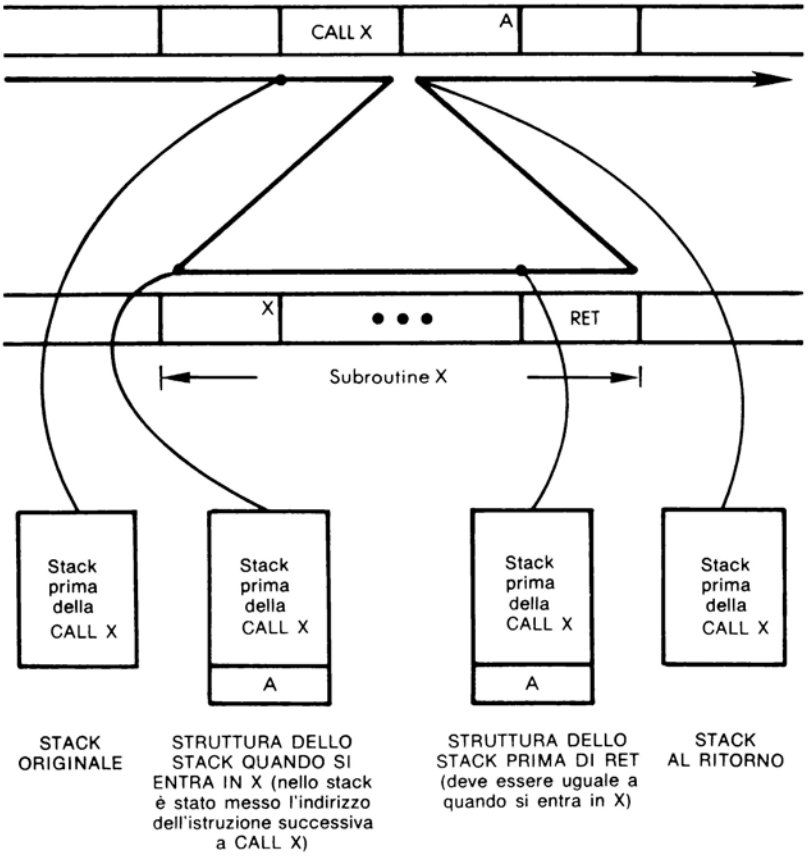
Si suppone che LENGTH prelevi in R1 l'indirizzo di una stringa di byte che termina con zero e restituisca in R0 il numero di byte, di questa stringa, diversi da zero. Per primo, vediamo il modo diretto:

```
LENGTH:   CLR R0          !Inizia a contare i byte!
COUNTLP:  TESTB @ R1      !Controlla il byte successivo!
          JR @ Z,ZERBYT   !Se è zero, abbiamo finito!
          INC R0          !Conta, ancora una altro byte diverso da
                           zero!
          INC R1          !Punta al prossimo byte!
          JR COUNTLP
ZERBYT:   RET
```

Quanto è riportato qui sopra illustra la forma di una subroutine. La prima istruzione che deve essere eseguita è accompagnata dalla label che deve essere usata per chiamare la subroutine (esempio nel nostro programma diciamo CALL LENGTH. L'istruzione RET completa le operazioni eseguite dalle subroutine ed esegue il ritorno al programma chiamante.

Lo Z8000 supporta, con il registro di stack, l'operazione delle istruzioni CALL e RET (vedasi Figura 3.4). Questo registro lo chiameremo SR – nel modo non segmentato è R15 ed in quello segmentato è RR14 – in funzione del modo normale o di quello di sistema esistono coppie diverse del registro di stack.

Quando viene eseguita la CALL, l'indirizzo della prima istruzione che segue la chiamata viene posto in SR prima di cedere il controllo alla subroutine. Perciò, quando nella Figura 3.2 viene eseguita l'istruzione CALL LENGTH, l'indirizzo della istruzione LD R3,R0 viene posto in SR, poi viene ceduto il controllo alla subroutine LENGTH.



La linea scura mostra il flusso del controllo; esempio l'ordine di esecuzione delle istruzioni.

Figura 3.4 — Uso dello Stack per le Istruzioni CALL e RET

Quando viene eseguita l'istruzione RET, l'indirizzo posto in cima allo stack (una parola nel modo non-segmentato, due nel modo segmentato) viene prelevato e messo nel PC. Naturalmente, per far sì che tutto funzioni, prima di eseguire la RET, la cima dello stack SR deve essere esattamente come era dopo aver appena eseguito la CALL. (Vedasi Figura 3.4).

Non ci dovrebbero essere problemi, se SR non viene utilizzato dalla subroutine o da qualche altra subroutine chiamata all'interno di questa. Ma, come vedremo in seguito, lo stack costituisce un mezzo conveniente di memorizzazione temporanea: ogni subroutine che usa lo stack deve garantire che esegue tanti POP (prelievi da stack) quanti PUSH (depositi

nello stack) fatti all'interno della subroutine stessa prima di effettuare il ritorno. In effetti, è anche più complicato di quanto abbiamo visto, ma vedremo come tenere sotto controllo questo problema nel capitolo dedicato alle Tecniche Avanzate di Programmazione.

Riguardando alla nostra routine LENGTH, notate i simboli COUNTLP e ZERBYT. Come funzioni sono molto simili a LOOP e DONE del programma principale. In effetti, avremmo dovuto chiamarle con gli stessi nomi, ma questo avrebbe confuso l'assemblatore quando, come viene normalmente fatto, avremmo cercato di assemblare il programma principale insieme alle subroutine.

Il problema sta nel fatto che tutti questi simboli sono delle *global*. Per cui se, per esempio, noi avessimo scritto JR LOOP nella subroutine LENGTH, quando avremmo incontrato JR COUNTLP, l'assemblatore avrebbe generato un salto, al di fuori della subroutine LENGTH, alla istruzione del programma principale con la label LOPP, con risultati molto indesiderati quando il codice fosse stato eseguito.

Adesso non vogliamo studiare ulteriormente i simboli *local* e *global*. Un loro trattamento è legato agli specifici sistemi e assemblatori che verranno discussi ulteriormente nel Capitolo X.

Ritorniamo indietro alla subroutine LENGTH per esaminarne la struttura generale. Essa è costituita da sette istruzioni. La prima e l'ultima sono eseguite una sola volta ciascuna: le rimanenti cinque costituiscono un loop. Nel loop, due istruzioni (TESTB @R1 e INC R0) determinano la funzione del loop e la sua chiusura.

L'istruzione CLR R0, che è nuova per noi, è semplicemente una alternativa alla LDK R0,#0. Inoltre, diversamente da LDK, può essere usata con un registro indice o indirizzo o col modo di *indirizzamento diretto*, che permette l'accesso ad una qualsiasi locazione di memoria.

Notate che nel loop, avremmo potuto voler piazzare l'istruzione INC R1 direttamente dopo TESTB @R1 per rendere ben chiaro a che cosa serviva R1 e come veniva usato. Sfortunatamente, questo avrebbe influenzato i bit di condizione per cui JR Z,ZERBYT non avrebbe controllato più a lungo lo stato del byte puntato da R1. Sarebbe stato possibile avere un modo per evitare questo se INC R1 potesse veramente essere combinato con il TESTB @R1 — si avrebbe avuto così un nuovo modo di indirizzamento che preleva per mezzo di un puntatore e lo incrementa in modo da essere sempre posizionato per il prossimo prelevamento.

Un modo di indirizzamento di questo tipo, chiamato *autoincremento*, esiste ma non è disponibile con l'istruzione di TEST. In verità, l'autoincremento e il suo cugino *autodecremento*, sono disponibili solamente per un

gruppo di istruzioni chiamate istruzioni di trasferimento di blocco e trattamento stringa (e implicitamente in PUSH, POP, ecc.).

Ogni calcolatore è progettato per fare particolarmente bene qualche cosa. Quando si scrive una routine come LENGTH, potenzialmente di utilità generale, è utile individuare delle caratteristiche del calcolatore che potrebbe ottimizzare i blocchi di codici eseguiti frequentemente come il loop di LENGTH.

TEST non ha alcuna relazione con il gruppo di istruzioni di trasferimento blocco ed elaborazione stringa, però una istruzione molto simile chiamata CP (confronto) ha una versione che opera sul blocco. La relazione tra CP e TEST è semplice: TESTB @ R1 è (quasi) lo stesso di CPB @ R1,#0, perciò controllare qualche cosa è analogo a confrontarlo a zero.

Ora, l'istruzione che sembra più promettente per il confronto di un blocco è CPIRB (confronta, incrementa, ripeti – ed opera sul byte), ma non è stata progettata esattamente per il nostro caso. Vi permette di esaminare una stampa, controllando byte per byte fino a che viene incontrata una determinata condizione o fino a quando è stato controllato un certo numero di byte. Perciò, è adatta quando conoscete la lunghezza della stringa ma non sapete se sarà sempre soddisfatta la condizione. Il nostro caso è l'opposto, noi non conosciamo la lunghezza della stringa, ma sappiamo che alla fine c'è uno zero.

Dopo questo preambolo, riportiamo la seconda versione di LENGTH:

```
LENGTH: CLR R0                  !Conta i byte in R0!
        CLRB RL2                !Controlla se c'è il byte zero!
        CPIRB RL2, @ R1,R0,EQ   !Scorre la stringa fino a zero!
        INC  R0                 !Compensazione per il byte zero!
        NEG R0                  !CPIRB conta all'indietro!
        RET
```

Questo programma rappresenta un fenomeno interessante: è molto breve (sei linee), il controllo passa in modo diretto dalla prima all'ultima istruzione, esiste un utile commento su ogni linea non ovvia, ed è stato preceduto da un buon trattamento dell'argomento – ma nonostante questo è ancora completamente incomprensibile. Non potrete comprendere questo programma senza analizzarlo attentamente per un po' di tempo e questa situazione non sarà cambiata da nessun commento o «documentazione» esterni.

L'istruzione CPIRB RL2, @R1,R0,EQ funziona in questo modo:

(1) Confronta il byte puntato da R1 con il byte zero in RL2.

(2) Per puntare al prossimo byte incrementa R1 di 1.

(3) Decrementa R0 di 1 (siccome R0 inizia da zero, la prima volta sarà uguale a −1, la seconda volta a −2, e così via).
(4) L'istruzione termina se R0 è zero (non lo sarà mai a meno che si abbia una stringa *molto* lunga) o se il risultato del confronto del passo uno rappresenta una eguaglianza (esempio era l'ultimo byte di chiusura); altrimenti si ritorna al passo (1).

La ragione per cui c'è l'istruzione INC R0 è la seguente: noi vogliamo contare solamente i byte diversi da zero, ma R0 viene decrementato solo quando si trova il byte zero.

ESERCIZIO 4: Avremmo potuto scrivere NEG R0; DEC R0 invece di INC R0; NEG R0. Perchè sono la stessa cosa? Qual è il più chiaro per il lettore?

Confrontiamo, adesso, le nostre due versioni di LENGTH in relazione allo spazio di memoria usato e al tempo di esecuzione. Nel prossimo capitolo impareremo come determinare il numero di byte di memoria e il numero di cicli di clock di esecuzione richiesti da ciascuna istruzione. La Figura 3.6 illustra questa analisi per il corpo principale delle nostre due ver-

	Byte	Cicli
CLR R0	2	7
TESTB @R1	2	8*(L + 1)
JR Z,ZERBYT	2	6*(L + 1)
INC R0	2	4*L
INC R1	2	4*L
JR COUNTLP	2	6*L
	12	21 + 28*L
CLR R0	2	7
CLRB RL2	2	7
CPIRB RL2,@R1,R0,EQ	4	11 + 9*(L + 1)
INC R0	2	4
NEG R0	2	7
	12	45 + 9*L

L = numero di byte della stringa diversi da zero

Figura 3.6 — Confronto delle Due Versioni di LENGTH

sioni della routine LENGTH. In nessuno dei casi viene presa in considerazione l'istruzione CALL e RET. La costante L rappresenta il numero di byte, diversi da zero, presenti nella stringa. Le due routine occupano lo stesso spazio di memoria. La nostra prima versione (diretta) è più veloce per stringhe di lunghezza zero od uno; per qualsiasi valore di L maggiore di uno, è molto più veloce la seconda versione.

ESERCIZIO 5: Per la chiave «abracadabra», quanti cicli vengono utilizzati da ciascuna versione? Quanti microsecondi, se il clock gira alla massima frequenza di 4 MHz (250 ns per ciclo)?

La seconda versione di LENGTH ha come *effetto collaterale* quello di cambiare i registri R1 e R2. Questo è sconveniente se LENGTH deve essere una routine di uso non specializzato. Onde evitare questo, possiamo usare lo stack per salvare, in entrata, i valori di questi due registri e per *ristabilirli* appena prima di uscire dalla routine. In Figura 3.7 è riportata la versione finale di LENGTH comprendente questa modifica.

```
!Subroutine per il calcolo delle lunghezze

  CALL LENGTH; R1 = indirizzo della stringa terminata con zero.
  Ritorna dalla routine con R0 = numero di byte diversi da zero presenti nella
  stringa – tutti gli altri registri sono inalterati.

  Assegnazione dei registri:
    R0: conta i byte diversi da zero
    R1: puntatore della stringa
    R2: RL2 contiene il byte zero per l'istruzione di confronto!

LENGTH:  PUSH @SR,R1          !Salva i registri!
         PUSH @SR,R2

         CLR R0               !Conta i byte in R0!
         CLRB RL2             !Byte zero per il confronto!
         CPIRB RL2,@R1,R0,EQ  !Scandisce per trovare lo zero!
         NEG R0               !CPIRB conta all'indietro!
         DEC R0               !Non conta il byte zero!

         POP R2,@SR           !Ristabilisce i contenuti dei registri!
         POP R1,@SR

         RET
```

Figura 3.7 — Subroutine LENGTH per il Programma di Crittografia

ESERCIZIO 6:

(a) I PUSH e POP addizionali aggiungono 34 cicli al tempo di esecuzione della routine. Quanti byte diversi da zero deve contenere una stringa affinchè quest'ultima versione sia più veloce della prima?

(b) I POP alla fine sono ordinati in modo opposto ai PUSH posti all'inizio. Che cosa significa?

(c) Se iniziassimo con PUSHL @SR,RR0 e finissimo con POPL RR0, @SR potremmo ridurre il tempo di esecuzione di 10 cicli. L'effetto è lo stesso? Qual è la versione più chiara?

Esaminiamo ora la subroutine MOD23 che calcola

(contenuti di R2) mod (contenuti di R3)

e pone il risultato in R2. In Figura 3.8 è illustrata questa subroutine. Le istruzioni SRLL e DIV sono le novità che troviamo in essa.

Il cuore della routine MOD23 è l'istruzione

DIV RR2,R4

Questa istruzione fa sì che il dato di 32-bit contenuto nel registro a parola lunga RR2 (costituito dalla parte più significativa R2 e meno significati-

```
!Subroutine MOD23

  CALL MOD23; R2 contiene I
              R3 contiene N ≠ 0
  Ritorna dalla subroutine con (I mod N) in R2, gli altri registri non sono mo-
  dificati.

  Assegnazione dei registri:
    RR2: Versione a parola-lunga del dividendo
    R4: Contiene temporaneamente il divisore (N)!

MOD23:  PUSH @ SR,R4   !Salva il contenuto del registro temporaneo!
        LD R4,R3       !Il divisore viene posto in R4!
        SRLL RR2,#16   !Sposta I nella parte meno significativa!
        DIV RR2,R4     !(I mod N) va in R2; il quoziente in R3!
        LD R3,R4       !Riporta N in R3!
        POP R4,@ SR    !Ristabilisce il contenuto del registro temporaneo!
        RET
```

Figura 3.8 — Subroutine MOD23 per il Programma di Crittografia

va R3) sia diviso per il dato a 16-bit contenuto in R4. Il quoziente è lasciato in R3 ed il resto è posto in R2.

Essendo questa la funzione mod, siamo solamente interessati al resto — che viene lasciato esattamente dove lo vogliamo. Il divisore, N, deve essere messo in R4 per eseguire la divisione poi, dopo la divisione deve essere riposto in R3, perchè si è supposto che la routine lo lasci in questo registro. Le istruzioni di PUSH e POP di R4 vengono eseguite per evitare che la routine abbia effetti indesiderati sui registri.

L'istruzione:

```
SRLL RR2,#16
```

è stata scritta per convertire il dato a 16-bit, contenuto in R2, in un dato a 32-bit posto in RR2 e con la parte più significativa della parola a zero. È uno *spostamento logico a destra* di 16-bit del registro lungo RR2.

Lo Z8000 è dotato di un certo numero di *spostamenti logici e aritmetici* a sinistra e a destra e operazioni di *rotazione*: su byte, parole o parole lunghe o combinazioni di una qualsiasi di queste con il bit di stato C. L'idea generale è che ciascun bit si muova di un numero specificato di posti (16 nell'esempio sopra riportato) verso destra o sinistra; l'unica differenza sta in ciò che accade alle estremità. Questo verrà spiegato in dettaglio nel prossimo capitolo. Nel nostro caso (uno spostamento logico a destra) i bit spinti fuori dall'estremo destro vengono persi e le corrispondenti posizioni vacanti sull'estremità sinistra vengono riempite con zeri.

Perciò, SRLL RR2,#6 è equivalente a

```
LD R3,R2
CLR R2
```

Avevamo iniziato il nostro programma di crittografia considerando un *programma principale*, cioè, un programma che non è una subroutine di un qualsiasi altro programma. In effetti, sarebbe più utile se lo si potesse usare come subroutine, per cui ora terminiamo l'esempio presentando, senza molto commento, una versione a subroutine del programma di crittografia.

In Figura 3.9 viene riportata questa routine. Si notino i cambiamenti che sono stati fatti. Per prima cosa, i registri sono stati salvati e ristabiliti utilizzando il comando LDM (caricamento multiplo), che è stato istituito per spostare numerosi blocchi di registri adiacenti in locazioni di memoria adiacenti.

ESERCIZIO 7: Perchè c'è differenza tra il 10 nelle istruzioni INC e DEC e 5 in LDM?

!Subroutine per Crittografare — continuazione dell'algoritmo E.

CALL ENCRYPT; R0 = indirizzo del testo
R1 = indirizzo della chiave
(il testo e la chiave sono stringhe terminate con zero)

Ritorna dalla subroutine con il testo sostituito dal testo crittografato, ed i registri inalterati.

Assegnazione dei registri:

R0 (RL0 & RH0): Registri di lavoro
R1: Indirizzo della chiave
R2: Indice I per posizionarsi nella chiave ($0 \leq I < N$)
R3: N (lunghezza della chiave)
R4: Indirizzo del testo da crittografare!

```
ENCRYPT:  DEC SR,#10          !Salva R0 su SR dello stack via R4!
          LDM @ SR,R0,#5
          LD R4,R0            !Puntatore del testo!

          CALL LENGTH         !Calcola N!
          LD R3,R0

          CLR R2              !Inizializza I!

LOOP:     LDB RL0,@ R4        !Byte successivo del testo!
          TESTB RL0           !Terminazione 0!
          JR Z,DONE

          LDB RH0,R1(R2)      !Byte I-esimo della chiave!
          XORB RL0,RH0        !Testo crittografato!

          LDB @ R4,RL0        !Sostituisci il testo, punta al successivo!
          INC R4

          INC R2              !Sostituisci I con (I + 1) mod N!
          CALL MOD23
          JR LOOP

DONE:     LDM R0,@ SR,#5      !Sostituisci i contenuti dei registri!
          INC SR,#10
          RET
```

Figura 3.9 — Versione a Subroutine del Programma di Crittografia

In secondo luogo, si noti l'istruzione

LDB RH0,R1(R2)

Questo è il modo di indirizzamento indicizzato mediante base – una delle caratteristiche rilevanti della struttura dello Z8000. Sfortunatamente non è disponibile con l'istruzione XOR – oppure avremmo potuto scrivere:

XORB RL0,R1(R2)

senza usare il passo intermedio per il caricamento di RH0.

ESERCIZIO 8: Quali altri cambiamenti notate? Li potete spiegare?

Con questo abbiamo terminato il nostro esempio. Siamo arrivati al programma finale analizzando il programma per crittografare fin dalle istruzioni iniziali. In questa analisi abbiamo utilizzato molte istruzioni e modi di indirizzamento dello Z8000. Nei prossimi due capitoli saranno presentate, in modo più organico, tutte le istruzioni e i modi di indirizzamento.

Capitolo IV

Istruzioni dello Z8000

PARTE 1.

Nei due precedenti capitoli abbiamo discusso l'architettura dello Z8000 ed abbiamo iniziato ad esplorare alcuni modi per usarlo. In questo capitolo e nel capitolo V daremo una presentazione sistematica delle istruzioni e dei modi di indirizzamento. Il nucleo di questo materiale sarà la descrizione delle 45 istruzioni che costituisce la Parte 2 di questo capitolo; in questa saranno dati i riferimenti fondamentali per le operazioni ed il formato delle istruzioni dello Z8000. Iniziamo con uno sguardo generale all'argomento.

Raggruppiamo in otto diverse categorie le istruzioni dello Z8000. Questa suddivisione è una supersemplificazione, dal momento che alcune istruzioni non appartengono chiaramente ad una categoria e quindi il numero delle categorie potrebbe essere maggiore o minore rispetto a quello da noi scelto.

Le categorie scelte per questa presentazione sono:

1. Operazioni e Controlli sui Dati
2. Trasferimenti dati
3. Predisposizione dei puntatori
4. Trasferimento del controllo
5. Ingresso/uscita
6. Controllo della CPU
7. Operazioni sui blocchi
8. Sincronizzazione multi-micro

Quando analizzeremo queste categorie elencheremo le istruzioni che vi appartengono. I mnemonici delle istruzioni sono elencati con le varie opzioni racchiuse in parentesi. Per esempio: ADC (B) rappresenta ADC e ADCB; ($\frac{D}{E}$)I rappresenta DI ed EI. In generale, quando una lettera è in

parentesi, quella lettera appare in alcune versioni dell'istruzione ma non in altre (esempio ADC, ADCB). Quando due o più lettere appaiono in parentesi, allora una di queste lettere appare in ogni versione data (esempio DI, EI). L'unica eccezione a tutto questo è $\binom{B}{L}$, in cui si può avere B, L o niente. Perciò LD $\binom{B}{L}$ rappresenta LD, LDB e LDL; R $\binom{L}{R}$ (C) (B) rappresenta RL, RLB, RLC, RLCB, RR, RRB, RRC, RRCB. Questa semplice abbreviazione ci permette di trattare, in modo unificato, le possibili variazioni di alcune istruzioni. Useremo questa annotazione in modo esteso in tutto questo capitolo; per cui dovreste essere sicuri di averlo capito.

Operazioni e controllo di dati

Questa categoria è costituita dalle istruzioni che modificano o controllano (test), il valore dei loro argomenti; sono le sole per le quali i bit di FLAGS vengono posizionati per indicare il risultato. (Le operazioni su blocchi e le istruzioni di sincronizzazione multi - micro usano i bit del FLAGS (indicatore) in modo speciale e alcune istruzioni permettono di manipolare o cambiare in modo esplicito questi bit in relazione alla variazione dello stato della CPU).

a) *Istruzioni aritmetiche*: ADD $\binom{B}{L}$, SUB $\binom{B}{L}$, MULT (L), DIV (L), ADC (B), SBC (B), NEG (B), DAB. Queste istruzioni implementano le funzioni aritmetiche standard di addizione, sottrazione, moltiplicazione e divisione. Sono le sole istruzioni di questa categoria che usano C per indicare il risultato. Nello Z8000, C viene usato principalmente per permettere l'implementazione delle versioni delle istruzioni aritmetiche in precisione multipla.

b) *Istruzioni logiche*: AND (B), OR (B), XOR (B), COM (B). Nelle «tabelle della verità» della figura 4.2 viene mostrata l'operazione di AND, OR, XOR e COM su argomenti ad un solo bit. Per argomenti di 8 bit o 16 bit ognuno dei bit viene gestito, indipendentemente, usando le regole del singolo bit.

Per esempio:

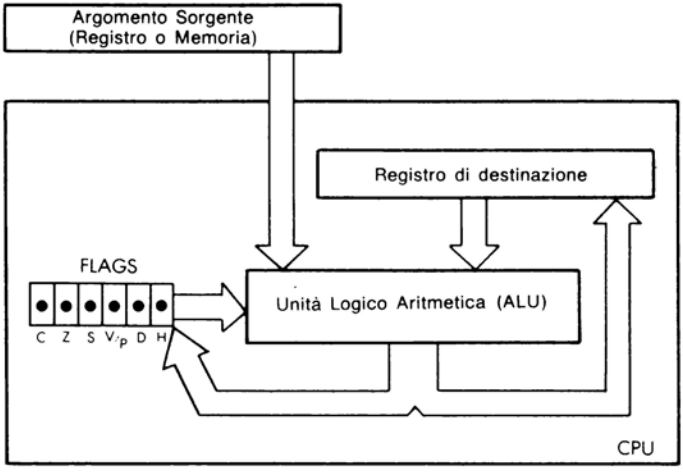
$$(11001110_2 \text{ ANDB } 01100101_2) = 01000100_2$$

AND, OR e XOR sono operatori a due argomenti; COM opera solo su un unico argomento. La versione a byte di queste istruzioni (e TESTB) sono le sole che usano P per indicare la parità del risultato.

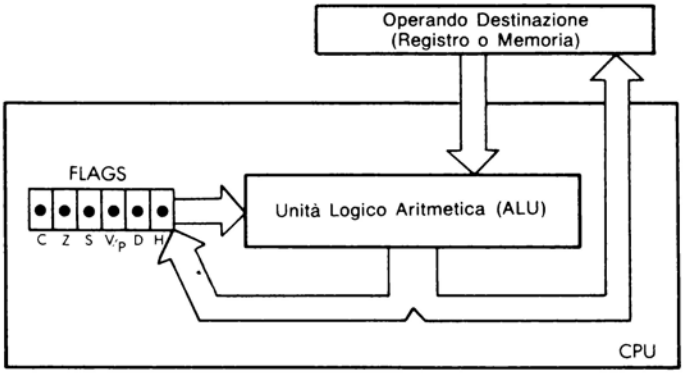
Le illustrazioni di Figura 4.1 mostrano, in modo schematico, come le istruzioni aritmetiche, logiche e rotazione/spostamento operino sui loro operandi all'interno della CPU. Ogni CPU possiede circuiti che permetto-

no di realizzare operazioni aritmetiche e logiche; questi circuiti sono generalmente indicati in un unico modo come unità logico e aritmetica (ALU).

Per l'istruzione di ADD, SUB, ecc., l'ALU prende tre ingressi: il registro di FLAGS, l'operando sorgente (sia dalla memoria che da un registro) e il registro destinazione. Il risultato viene posto nel registro destinazione e vengono fissati i nuovi valori di FLAGS.



Percorso dati per ADD, SUB, MULT, DIV, AND, OR, XOR, spostamento, rotazione, e per ADC, SBC, rotazione di cifra (solamente registri sorgente).



Percorso dati per NEG e COM; e per DAB (solamente registro)

Figura 4.1 — Percorso Dati per Istruzioni Aritmetiche, Logiche e Spostamenti/Rotazioni.

Useremo illustrazioni simili, qualche volta senza ulteriori commenti, per tutte le categorie di istruzioni che abbiamo stabilito per lo Z8000.

c) *Istruzioni di spostamento/rotazione*: R ($^{L}_{R}$) (C) (B), S ($^{D}_{L}$) ($^{A}_{L}$) ($^{B}_{L}$), R ($^{L}_{R}$) DB. Queste istruzioni permettono di spostare i bit di un registro verso sinistra o destra o circolarmente. Tutto questo viene illustrato in Figura 4.3.

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

COM	0	1
	1	0

Figura 4.2 — Tabella della Verità per Operazioni Logiche.

Le rotazioni possono essere pari ad una o due posizioni e possono includere od escludere C (sebbene C faccia ancora parte dello spostamento, anche se escluso dalla rotazione).

L'entità degli spostamenti è espressa da un numero compreso tra 0 ed il valore di bit dell'argomento. Il conteggio dello spostamento può essere specificato esplicitamente nelle istruzioni o può essere preso da un registro al momento dell'esecuzione.

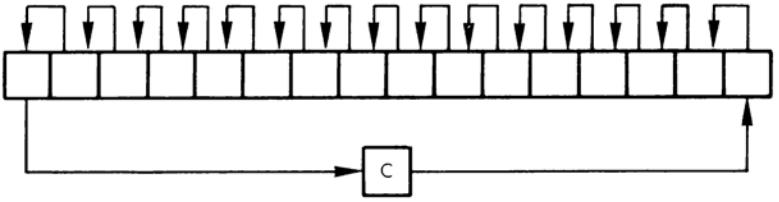
La R ($^{L}_{R}$) DB è un'istruzione specializzata, utilizzata per muovere circolarmente delle cifre mediante due registri di un byte ciascuno. Questa ultima istruzione è illustrata in Figura 4.4.

d) *Istruzioni per modificare il contatore/puntatore*: DEC (B), INC (B). Queste istruzioni permettono che i loro argomenti siano incrementati o decrementati di una quantità compresa fra 1 e 16; differiscono dalle corrispondenti situazioni di somma e sottrazione per il fatto che C non viene alterato e V viene gestito diversamente.

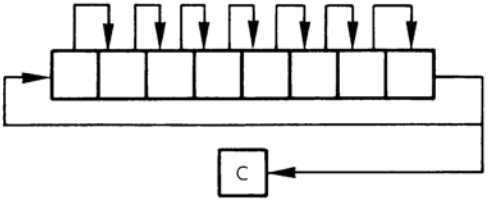
e) *Istruzioni di test*: BIT (B), CP ($^{B}_{L}$), TEST ($^{B}_{L}$), TSET (B). Queste istruzioni posizionano i bit di FLAGS dopo aver esaminato (invece di eseguire operazioni) i propri argomenti. BIT (B) permette di esaminare uno qualsiasi dei bit di un argomento costituito da 8 o 16 bit e posiziona Z per comunicare il risultato. Il numero di bit che devono essere controllati possono essere contenuti nell'istruzione o possono essere prelevati da un registro al momento dell'esecuzione.

CP ($^{B}_{L}$) è l'istruzione base di confronto e TEST ($^{B}_{L}$) può quasi essere considerato come un caso speciale di CP ($^{B}_{L}$) se si suppone che uno degli argomenti di confronto sia uguale a 0. L'idea è che le sequenze di codice del tipo

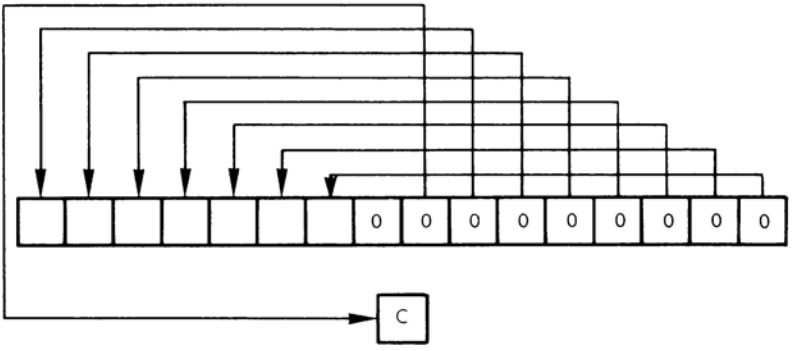
```
CP X, Y; JR GT, ABC
TEST X; RET MI
```



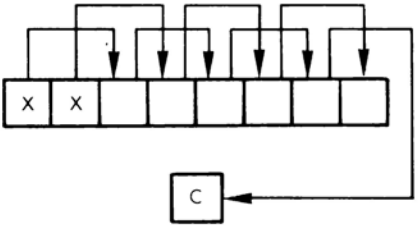
Rotazione a sinistra di un registro a 16-bit comprendente C



Rotazione a destra di un registro di 8-bit non comprendente C



Spostamento di 9 posizioni a sinistra di un registro a 16-bit



Spostamento di 2 posizioni a destra di un registro di 8 bit
x = 0 o 1 in funzione del fatto che lo spostamento sia aritmetico o logico e, se è aritmetico, dal valore originale del bit più a sinistra

Figura 4.3 — Alcuni Spostamenti e Rotazioni

siano facilmente ricordabili. Le due linee sopra riportate significano:

«se x > y, allora salta ad ABC»

«se x < 0, allora esegui il ritorno»

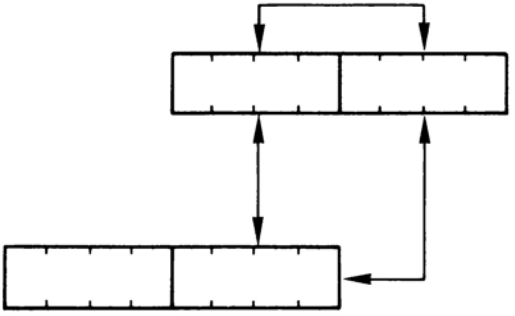
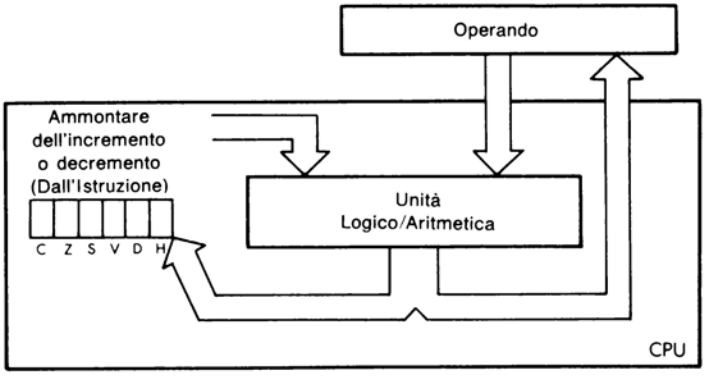


Figura 4.4 — La Circolazione di Digit Realizzata con R ($\frac{L}{R}$) DB.

Questo si realizza assegnando dei codici mnemonici del tipo GT, LT, PL, MI, ecc. alle diverse combinazioni dei bit di FLAGS. Esistono 16 tipi diversi di queste combinazioni che possono essere rappresentate in un campo di 4 bit presente nelle istruzioni condizionali (TCC (B), JP, JR, RET) e nel confronto di un blocco CP (S) ($\frac{D}{I}$) (R) (B). Questi nomi mnemonici appariranno un po' più avanti in questo capitolo. Per adesso ricordatevi che se si scrive CP x, y, allora x sta sulla sinistra della relazione definita



Percorso Dati per INC e DEC

Figura 4.5 — Percorso Dati per le Istruzioni che Modificano il Contatore/Puntatore.

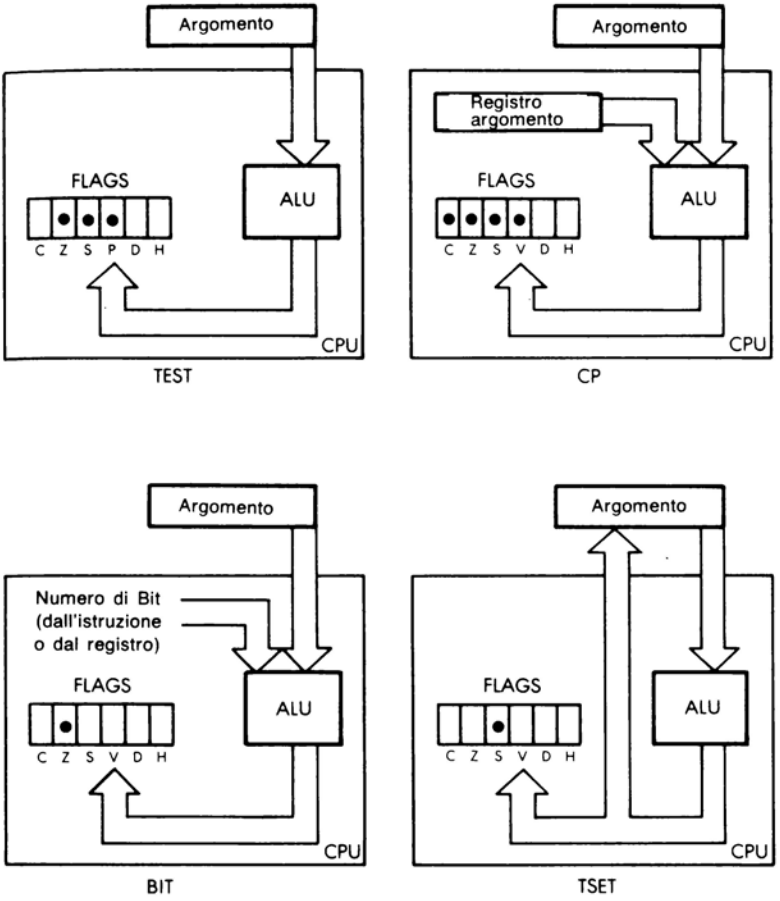


Figura 4.6 — Percorso Dati per le Istruzioni di Test

dal mnemonico (> nel campo sopra riportato) e y appare sulla destra. Per TEST x, la x sta a sinistra e 0 sulla destra.

L'istruzione TSET potrebbe essere posta in una categoria a parte, siccome combina una funzione di test e una di trasferimento dato. Il concetto consiste nel controllare un indicatore (flag) e cambiare contemporaneamente il suo valore portandolo ad uno stato «non significativo»: questo permette di avere una protezione contro due tests positivi, «simultanei», dell'indicatore da parte di processi concorrenti. TSET è definita per facilitare l'uso dei *semafori*; una tecnica di scienza dei calcolatori nata dallo studio dei processi concorrenti.

Trasferimento dati

Questa categoria è costituita da istruzioni che trasferiscono un dato da un luogo ad un altro. Il dato trasferito può essere implicito o contenuto nelle istruzioni oppure può essere nella locazione specificata dall'argomento dell'istruzione. I contenuti originali della destinazione del trasferimento sono irrilevanti e vengono persi (eccetto per EX(B)).

Le istruzioni di trasferimento dati lasciano inalterati tutti i bit di FLAGS.

a) *Istruzioni di trasferimento costanti*: CLR (B), LDK, RES (B), SET (B), TCC (B). Queste istruzioni trasferiscono il valore costante (fisso) specificato nell'istruzione nella locazione di destinazione. CLR (B) pone a zero il valore di destinazione. LDK contiene un campo di quattro bit per permettere di posizionare il valore di destinazione con un valore compreso tra 0 e 15. RES (B) e SET (B) permettono di porre a 0 o ad 1 uno specifico bit; come già visto per BIT (B), il numero che specifica il bit interessato può essere contenuto nell'istruzione o prelevato dal registro.

TCC (B) è un'istruzione SET (B) condizionale. Il numero di bit è sempre 0; esempio: viene influenzato il bit meno significativo dell'argomento. È importante non confondere questo trasferimento condizionale con il modo con cui i bit di FLAGS sono posti ad 1. Tutte le istruzioni che usano un bit di FLAGS per indicare una condizione pongono ad 1 il bit per indicare l'avvenuta condizione o azzerano il bit per indicare il contrario; TCC (B) pone ad 1 il bit se la combinazione specificata dei bit di FLAGS è vera e *non fa nulla* se la combinazione è falsa.

b) *Istruzioni di Trasferimento di Variabili:* LD $\binom{B}{L}$, LDR $\binom{B}{L}$, LDM, EX (B), EXTS $\binom{B}{L}$. Queste istruzioni muovono il dato, che si trova nel loro argomento sorgente, nel loro argomento di destinazione. LD $\binom{B}{L}$ è l'istruzione di base per il trasferimento dei dati. Utilizza tutti i modi di indirizzamento che ha lo Z8000 per indirizzare i *dati* o la *memoria di stack*. Solamente le istruzioni LD $\binom{B}{L}$ ed LDA possono usare i modi di indirizzamento tramite base ed indicizzato tramite base.

LDR$\binom{B}{L}$ è la sola istruzione che permette l'accesso alla *memoria di programma*. Utilizza l'indirizzamento relativo al PC e lo spiazzamento del PC è memorizzato nell'istruzione. Questo significa che non c'è convenienza nell'avere nel programma degli insiemi o tabelle, a meno che anche la memoria che essi occupano sia (possibilmente con un diverso insieme di indirizzi) parte della memoria dati. L'unica alternativa possibile è simulare uno schema di indirizzamento indicizzato e memorizzare il valore dello spiazzamento del PC risultante nel campo spiazzamento di un'istruzione LDR$\binom{B}{L}$; questo richiede la possibilità di scrivere nella memoria di

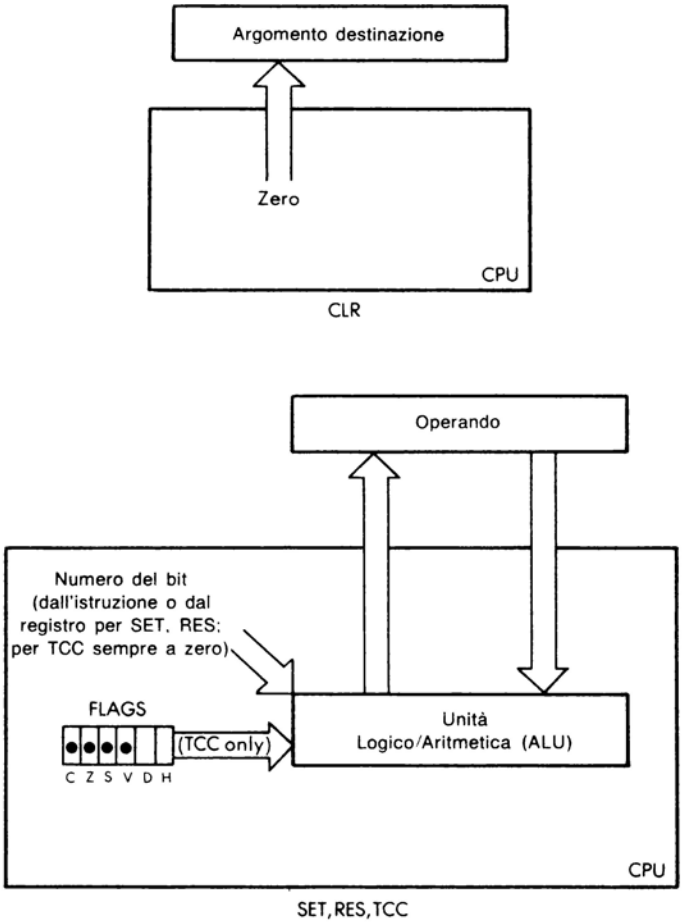


Figura 4.7 — Percorso Dati per le Istruzioni di Trasferimento Dati e Posizionamento del Puntatore Alcuni Trasferimenti di Costante.

programma. In ogni evento, LDR ($^{B}_{L}$) non permetterà dei riferimenti al di fuori del segmento corrente.

L'istruzione LDM permette, con un unico comando, di caricare da o verso la memoria da 1 a 16 registri. Questa prestazione è disponibile per facilitare commutazioni rapide di contesto in ambiente multiprogrammato. È anche utile per salvare e ristabilire i registri nelle subroutine di interruzione.

L'istruzione EX(B) permette di scambiare i contenuti dei registri e del-

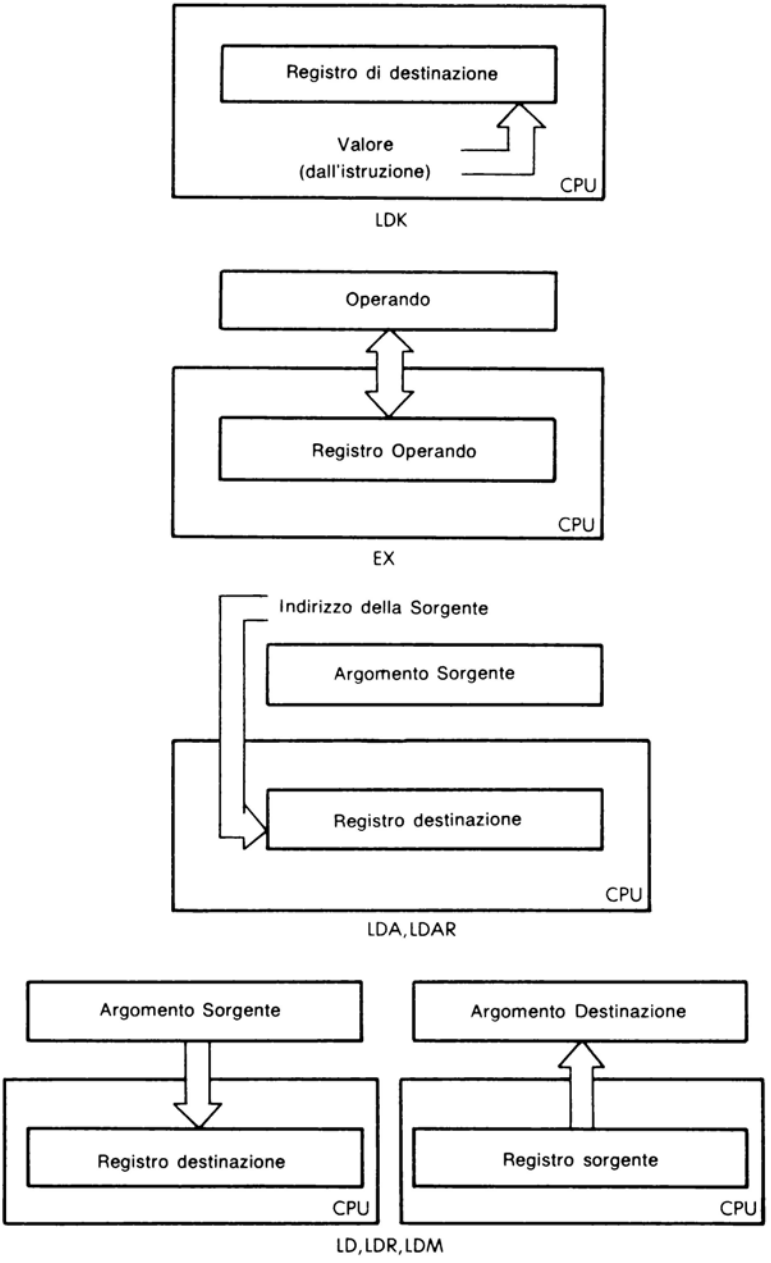


Figura 4.7 (b) — Percorso Dati per l'Istruzione mento di Trasferimento Dati e Posizionamento nbio e Puntatore — Scambio e Caricamento

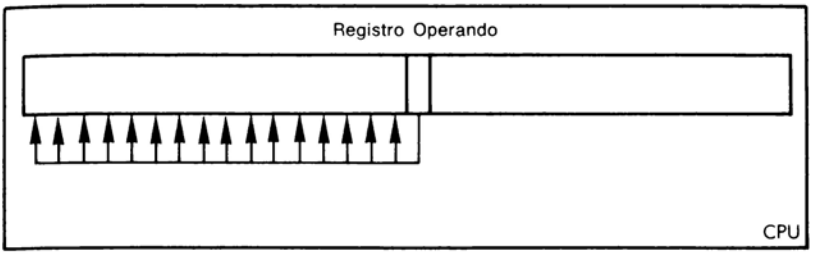
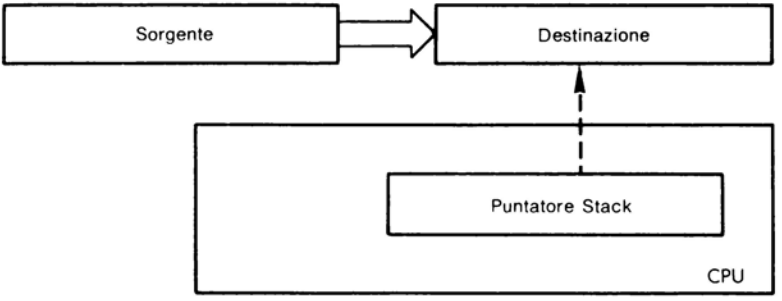
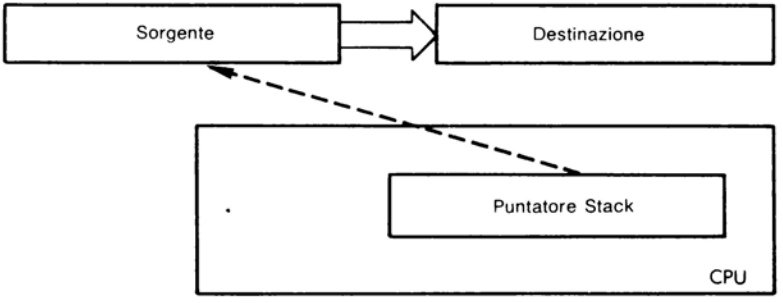


Figura 4.7 (c) — Percorso Dati per l'Istruzione di Trasferimento Dati e Posizionamento Puntatore — Estensione del Segno.



PUSH



POP

Figura 4.7 (d) — Percorso Dati per l'Istruzione di Trasferimento Dati e Posizionamento Puntatore — Trasferimenti su Stack.

la memoria. L'istruzione EXTS ($^{B}_{L}$) permette di espandere un argomento in un registro per poter occupare un registro più grande (senza cambiarne il valore) propagando il bit di segno del valore originale su tutto lo spazio del registro addizionale.

c) *Istruzioni di Trasferimento su Stack:* POP(L), PUSH(L). Sono operazioni di salvataggio o prelievo descritte nel Capitolo II (vedasi Figura 2.2). Non esistono versioni a byte di POP e PUSH, perché lo stack hardware (controllato da R15 o RR14) deve sempre essere pronto a lavorare con i 16 bit associati alle interruzioni ed alle chiamate di subroutine. Questo rende difficile salvare e ristabilire i registri a byte RH0, RL0, ...

Posizionamento del Puntatore

Esistono due istruzioni che caricano un registro indirizzo con *l'indirizzo* del loro argomento: LDA e LDAR. Queste istruzioni permettono, in una sola operazione, il caricamento di indirizzi segmentati e forniscono un'approcio uniforme che funzioni sia con indirizzi segmentati che con quelli non - segmentati.

Siccome gli indirizzi non dipendono dalla dimensione degli argomenti memorizzati ad uno specifico indirizzo, LDA X ed LDAR X possono essere utilizzate per ottenere l'indirizzo di X, non tenendo conto se X è un byte, una parola o una parola lunga. La dimensione del registro indirizzo usato è legata al funzionamento in modo segmentato o non-segmentato della CPU – questa informazione non è contenuta nell'istruzione.

Ci sono delle operazioni interne che, siccome funzionano indipendentemente dallo stato delle linee di uscita dal dispositivo, non fanno distinzione tra gli indirizzi dell'area programmi, dati e stack.

Le istruzioni di posizionamento del puntatore non alterano i bit di FLAGS.

Trasferimento del Controllo

Questa categoria è costituita da istruzioni che interrompono il flusso di controllo modificando il valore di PC. Eccetto per il salvataggio ed il ripristino dello stato della CPU associato alle interruzioni, nessuna di esse opera su FLAGS.

(a) *Salti:* JR, JP, D(B)JNZ.

(b) *Istruzioni per subroutine*: CALL, CALR, RET. Queste sono le istruzioni usate per realizzare le subroutine. CALL e CALR salvano l'indirizzo di ritorno nello stack (usando R15 o RR14) e RET lo ripristina in PC. (Vedasi Figura 3.4). CALR è il formato in parola-singola dell'istru-

zione CALL ed utilizza gli indirizzi, relativi a PC, in un campo di 4096 *parole* centrate (approssimativamente) sull'istruzionie CALR.

RET consente l'uso di un campo di codice condizione (descritto sopra nella «istruzione di test»). Questo può essere conveniente quando il ritorno da subroutine si verifica all'interno di un loop, ma dovrebbe essere usato con moderazione se origina delle subroutine contenenti più di una istruzione RET. Le routine contenenti più di una istruzione RET possono essere difficili da capire e da cambiare.

(c) *Istruzione di interruzione/trappole:* SC, IRET. SC è la «chiamata sistema» da parte di una trap. Fa si che il PC venga salvato nello stack, come per la CALL, insieme alla FCW e alla «ragione» (una copia dell'istruzione SC, un byte della quale è un indice ad una tabella di routine di sistema). IRET non fa niente di tutto questo, ma ripristina FCW e PC ed elimina la «ragione». (Vedasi Figure 2.8 e 2.9).

Non ci sono istruzioni esplicite per generare interruzioni esterne o trap da errori. Le interruzioni e le trap da errori determinano il salvataggio dello stato della CPU e della «ragione» proprio come SC ed utilizzano per il ritorno l'istruzione IRET.

Dal momento che IRET ricostituisce lo stato salvato per SC o altre interruzioni o trap, l'effetto finale è che non vengono cambiati i bit di FLAGS. Una possibile eccezione potrebbe essere una routine di sistema che modifica la copia di FLAGS contenuta nello stack in modo tale da restituire una informazione al chiamante. Nel Capitolo IX analizzeremo in maggior dettaglio come, utilizzando SC, vengano trasmessi i parametri verso e dalle routine di sistema.

Ci sono altri due punti interessanti riguardanti le trap: esiste una trap riservata che può essere usata se non avete bisogno di garantire la compatibilità dei vostri programmi con le versioni future dello Z8000; e ci sono alcune trap da errore che potrebbero essere molto utili se esistessero veramente. La trap riservata è «la trap per istruzione non implementata» che si verifica quando si usa una qualsiasi delle sei combinazioni dei campi codice operativo e modo. (Più avanti in questo capitolo discuteremo i campi di modo e codice operativo). Ciascuna di queste sei istruzioni potrebbe essere usata esattamente come viene usata SC — fornendo ognuno in tal modo altri sei gruppi di 256 routine di sistema.

Sarebbe utile avere molte trap da errore; in particolare nello Z8000 esistono due dimenticanze piuttosto notevoli riguardanti il tipo di disaccoppiamento: l'utilizzo di una istruzione a parola o parola-lunga con un indirizzo dispari e l'utilizzo di un numero di registro a parola multipla che superi quello legalmente ammesso. Per esempio, l'istruzione LD R0, @R1 richiede che gli indici di R1 siano pari, se sono dispari, il comportamento

dello Z8000 è *indefinito*. Non ci viene detto che cosa farà, ma per esempio potrebbe ignorare i bit meno significativi degli indirizzi sbagliati, non considerare nessuno di quelli che suppone siano degli zeri. Questa è una seria limitazione che dovrebbe essere eliminata nelle versioni future dello Z8000. Il formato della PSA potrebbe essere alterato o si potrebbe condividere una trap esistente, per permettere alla «ragione» di distinguere tra questa e il suo significato originale. Nel frattempo, l'utilizzatore dovrebbe realizzare tale trap utilizzando le uscite $B/\overline{W}$ e AD_0 della CPU per generare una interruzione non mascherabile. Questa soluzione presenta tre svantaggi principali: (1) essa rileva l'utilizzo sbagliato della parola/byte ma non quello relativo al registro; (2) non può rilevare i trasferimenti ad indirizzi dispari siccome il PC non memorizza (latch) il bit zero; esempio il bit zero del PC è sempre zero indipendentemente da ciò che la CPU cerca di memorizzarvi; (3) genera una interruzione *dopo* che l'istruzione ha completato la sua azione indefinita. (Vedasi Figura 9.14).

Ingresso/Uscita

Lo Z8000 riconosce uno spazio indirizzo di 65.536 byte per le sue operazioni di Ingresso/Uscita (I/O) ed un altro spazio indirizzi della stessa dimensione per le operazioni di I/O «speciali». Le operazioni speciali di I/O sono state definite per essere usate con la MMU, la quale non può avere le linee indirizzi/dati $AD_7 - AD_0$ disponibili per essa. Internamente alla CPU non esistono differenze tra gli I/O speciali ed ordinari.

L'istruzione di I/O è molto simile all'istruzione di caricamento ma è molto più limitata nei modi di indirizzamento. Ad eccezione dell'istruzione di I/O per blocco, da discutere più avanti, tutte le istruzioni di I/O si sviluppano tra un registro della CPU ed un indirizzo di I/O specificato nella istruzione o contenuto in un registro. Non è possibile indicizzare gli indirizzi di I/O.

L'istruzione I/O base,(S)$\binom{\text{IN}}{\text{OUT}}$(B)non altera i bit di FLAGS. È una istruzione privilegiata.

Controllo della CPU

Questa categoria di istruzioni ha più a che fare con la gestione della CPU che con il calcolo. I bit di FLAGS vengono posizioniati solamente in modo esplicito.

(a) *Gestione dei registri di controllo:* ( )I, LDPS, LDCTL. Queste istruzioni permettono di accedere ai registri di controllo della CPU. L'istruzione $\binom{D}{E}$)I permette la disabilitazione (mascherata) o l'abilitazione

(non mascherata) di qualsiasi combinazione delle interruzioni mascherabili. La LDPS carica FCW e PC dalla memoria dati. LDCTL è una istruzione di caricamento per i quattro registri di controllo: FCW, PSAP, NSP e REFRESH. Permette trasferimenti tra uno qualsiasi di questi registri e un generico registro.

Tutte queste sono istruzioni privilegiate.

(b) *Gestione di FLAGS*: COMFLG, RESFLG, SETFLG, LDCTLB. Le prime tre consentono di operare su qualsiasi combinazione dei bit C, Z, S, V/P. LDCTLB è una istruzione di caricamento dell'intero registro di FLAGS.

(c) *Sincronizzazione esterna*: HALT, NOP. L'istruzione HALT è molto più simile all'istruzione WAIT di quanto non lo sia rispetto all'HALT tradizionale. Determina la sospensione delle operazioni della CPU fino a quando non viene inviata una interruzione – IRET fa ripartire il programma all'istruzione successiva ad HALT. Inoltre mentre la CPU è in HALT continuano le funzioni vitali quali i cicli di rinfresco e il protocollo di acquisizione del bus.

L'istruzione NOP fa si che la CPU non faccia nulla durante la sua esecuzione (sette cicli). HALT è una istruzione privilegiata, NOP non lo è.

Operazioni di blocco

Ci sono quattro operazioni di blocco. Tre sono le versioni di blocco delle istruzioni sopra riportate; una invece non ha la sua controparte in quanto detto sopra.

La struttura generale di una istruzione che opera sul blocco richiede: un registro contatore e due registri puntatori. L'esecuzione dell'istruzione determina il decremento del registro contatore ed incrementa o decrementa uno od entrambi i registri puntatori. L'istruzione può essere posta insieme ad altre operazioni in un loop oppure può essere ripetuta automaticamente fino a quando il contatore diventa zero o (in alcuni casi) vengono verificate altre condizioni. Tutte le istruzioni di blocco possono essere interrotte durante l'esecuzione con una perdita di tempo pari a sette cicli per ogni interruzione. Nessuna altra istruzione dello Z8000 può essere interrotta.

Tutte le istruzioni di blocco usano l'indicatore V per segnalare quando il contatore è arrivato a zero. In tutte queste istruzioni Z viene posizionata o lasciata in uno stato indefinito.

La Figura 4.9 mostra i tre registri e la sequenza di operazioni eseguite nella versione di blocco delle istruzioni CP, LD, e I/O.

(a) *Test dei dati di un blocco*: CP (S) ($^{D}_{I}$) (R) (B). Una stringa di paro-

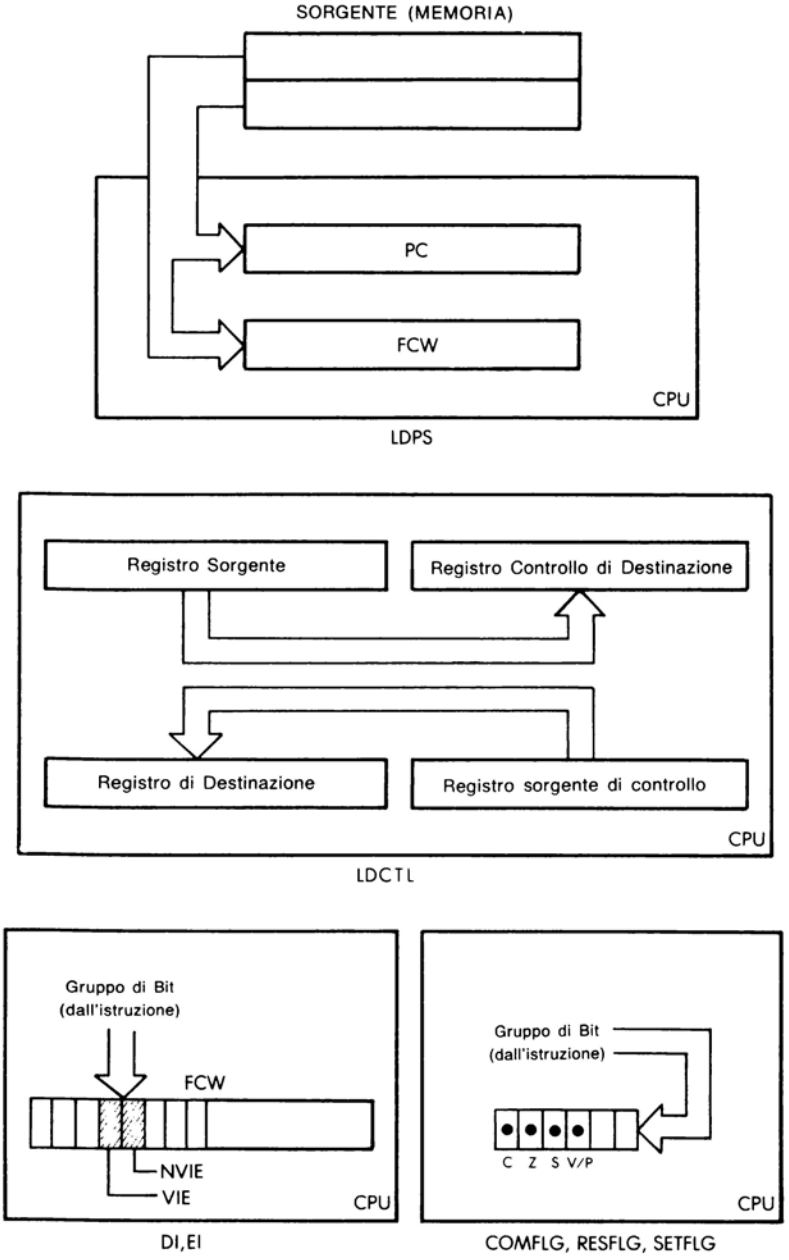


Figura 4.8 — Percorso Dati per le Istruzioni di Controllo della CPU

le o byte di dati viene confrontata con un'altra stringa oppure con un valore contenuto in un registro. Un campo di codice condizione nell'istruzione permette di specificare la combinazione dei bit di FLAGS che si sta cercando. Z viene posto ad uno se viene rilevata l'uguaglianza. Nel modo ripetitivo, l'istruzione viene ripetuta fino a quando V o Z è uguale ad uno. C viene lasciato indefinito.

(b) *Trasferimento di un blocco dati*: LD $\binom{D}{I}$ (R) (B). La stringa di parole o byte viene trasferita da un posto ad un altro. Se c'é una sovrapposizione delle stringhe è importante effettuare una scelta oculata tra modo autoincrementato ed autodecrementato.

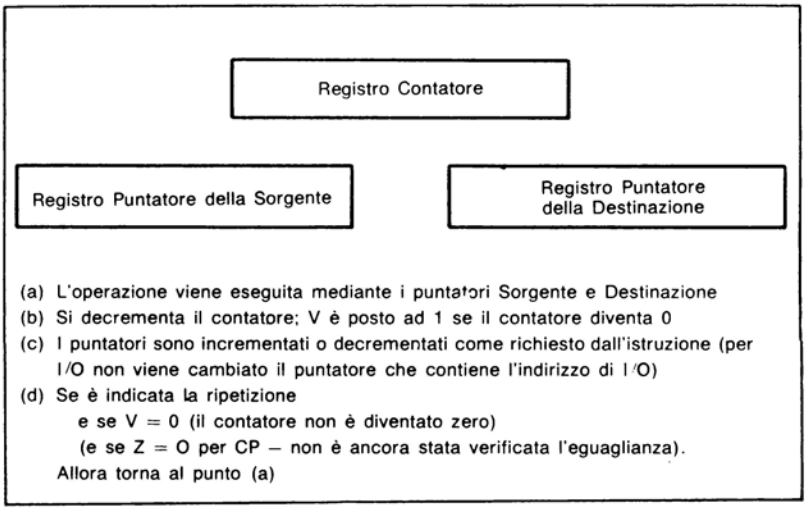


Figura 4.9 — Registri e Sequenza di Funzionamento per le Versioni a Blocco di CP, LD, I/O

(c) *Blocco di I/O:* (S) $\binom{IN}{OUT}$ $\binom{D}{I}$ (R) (B). Una stringa di parole o byte è inviata dalla memoria in uscita o ricevuta in ingresso ad un indirizzo costante di I/O specificato in un registro.

(d) *Conversione e test di un blocco*: TR (T) $\binom{D}{I}$ (R) B. Questa istruzione usa una tabella di conversione di 256 byte: la conversione di un generico valore i con $0 \leq i \leq 255$ è l'i-esimo byte della tabella di conversione.

Una stringa di byte viene convertita utilizzando una tabella di conversione il cui indirizzo è tenuto in un registro. Ciascun valore convertito sostituisce il valore originale oppure viene posto in un registro dedicato (RH1) e controllato per accertare se è uguale a zero. Usando l'opzione della ripetizione, l'operazione viene eseguita fino all'esaurimento del con-

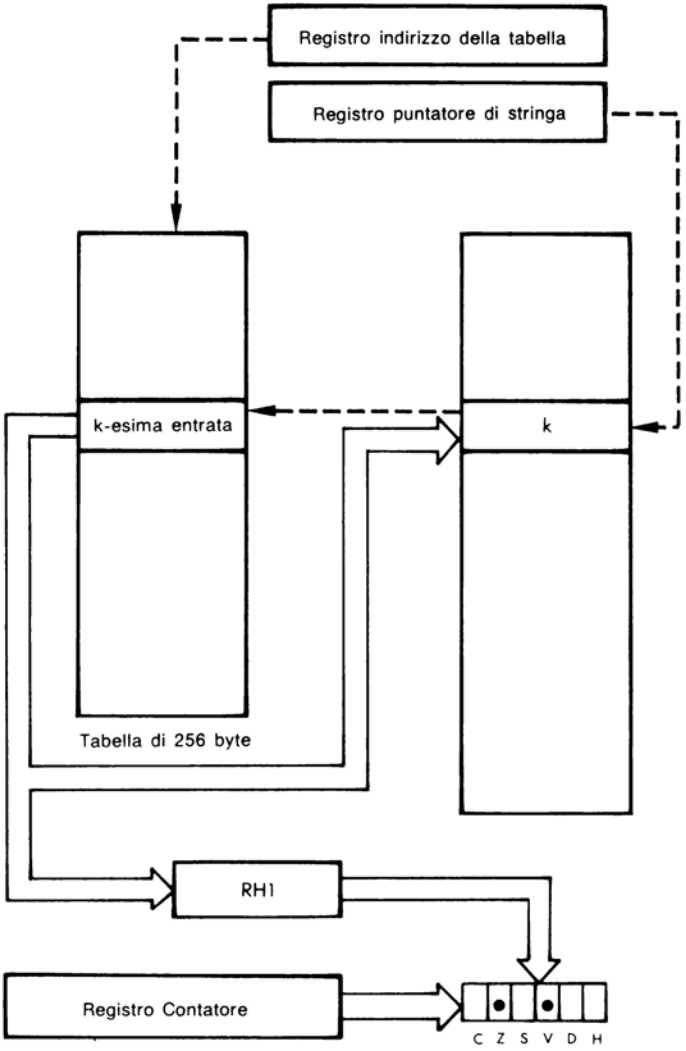


Figura 4.10 — Funzionamento delle Istruzioni di Conversione e di Test

teggio o (se c'è l'opzione test) fino a quando non viene trovato un valore di conversione diverso da zero. (Vedasi Figura 4.10).

Sincronizzazione Multi-Micro

Questa categoria di istruzioni si riferisce alla sincronizzazione degli accessi, da parte di CPU diverse ad una risorsa condivisa. Nell'architettura

del sistema esistono, a tale scopo, due piedini della CPU (uno di ingresso, uno di uscita) e quattro bit per il bus esterno. MREQ gestisce il protocollo associato a questa struttura. Gli indicatori Z ed S vengono usati in modo speciale per indicare l'avvenuta acquisizione del controllo della risorsa condivisa. MBIT controlla il piedino d'ingresso e segnala lo stato di questo utilizzando S e Z; MRES e MSET controllano il piedino di uscita lasciando inalterati i bit di FLAGS.

Tutte queste sono istruzioni privilegiate.

Adesso abbiamo terminato la nostra panoramica delle istruzioni dello Z8000. Nella Figura 4.11 è riportato un sommario delle categorie di raggruppamento delle istruzioni.

Codici Condizione

Nel precedente paragrafo di questo capitolo abbiamo accennato alle 16 combinazioni di codici condizione che possono essere codificati in un campo di quattro bit in alcune delle istruzioni condizionali.

Le 16 condizioni sono costituite da otto condizione base e dai loro apposti. Nella codifica dei quattro bit, il bit più significativo viene usato come indicatore di «opposto». Per esempio, se 1 è la codifica di «meno di» allora 9 è la codifica di «più grande di o uguale a».

Delle otto condizioni base, quattro sono le condizioni con bit singolo C = 1, Z = 1, S = 1, V/P = 1 ed una è la condizione di «sempre falso». Le tre condizioni rimanenti sono:

less than (LT): S = 1 o V = 1, ma non entrambi
less than o equal to (LE): come sopra o Z = 1
unsigned less then or equal to (ULT): C = 1 o Z = 1

N.d.T. less than = minore di; less than or equal to = minore di o uguale a; unsigned less than or equal to = valore assoluto minore di o uguale a.

Alle varie condizioni viene assegnato un nome simbolico in modo da poter avere un valore mnemonico da usare nell'istruzione di confronto CP. CP x,y posiziona i bit di FLAGS in funzione del risultato della sottrazione x − y; x è *minore di y* se e solo se x − y è negativo. Se non c'è riporto aritmetico, questo è analogo ad S = 1. Se invece c'è riporto aritmetico questo è analogo ad S = 0.

Analogamente x = y se e solo se x − y = 0, per cui la condizione Z = 1 viene definita come *uguaglianza*, inoltre Z = 1 combinata con la condizione «*minore di*» forma la condizione «*minore di o uguale a*».

Considerando x ed y come numeri del campo da 0 a $2^n - 1$ invece che

1. Operazioni e Controllo di Dati (si usa la FLAGS standard)
   (a) Aritmetiche: **ADD($^{B}_{L}$), SUB($^{B}_{L}$), MULT(L), DIV(L), ADC(B), SBC(B), NEG(B), DAB**
   (b) Logiche: **AND(B), OR(B), XOR(B), COM(B)**
   (c) Spostamento/Rotazione: **R($^{L}_{R}$)(C)(B), S($^{D}_{L}$)($^{A}_{L}$)($^{B}_{L}$), R($^{L}_{R}$)DB**
   (d) Modifica del Contatore/Puntatore: **DEC(B), INC(B)**
   (e) Test: **BIT(B), CP($^{B}_{L}$), TEST($^{B}_{L}$), TSET(B)**
2. Trasferimento Dati (i bit di FLAGS non vengono alterati)
   (a) Trasferimento di Costanti: **CLR(B), LDK, RES(B), SET(B), TCC(B)**
   (b) Trasferimento di Variabili: **LD($^{B}_{L}$), LDR($^{B}_{L}$), LDM, EX(B), EXTS($^{B}_{L}$)**
   (c) Trasferimento su Stack: **POP(L), PUSH(L)**
3. Posizionamento del Puntatore (i bit di FLAGS non vengono alterati): **LDA, LDAR.**
4. Trasferimento del Controllo (ad eccezione del salvataggio/ripristino non vengono alterati i bit di FLAGS)
   (a) Salti: **JR, IP, D(B)JNZ**
   (b) Subroutine: **CALL, CALR, RET**
   (c) Interruzione/Trappola: **SC, IRET***
5. Ingresso/Uscita (i bit di FLAGS non vengono alterati): **(S)($^{IN}_{OUT}$)(B)***.
6. Controllo della CPU (FLAGS viene alterato solo in modo esplicito)
   (a) Gestione dei Registri di controllo: **($^{D}_{E}$)I*, LDPS*, LDCTL***
   (b) Gestione di FLAGS: **COMFLG, RESFLG, SETFLG, LDCTLB**
   (c) Sincronizzazione esterna: **HALT*, NOP**
7. Operazioni di Blocco (uso speciale di Z e V)
   (a) Test Dati: **CP(S)($^{D}_{I}$)(R)(B)**
   (b) Trasferimento Dati: **LD($^{D}_{I}$)(R)(B)**
   (c) I/O: **(S)($^{IN}_{OUT}$)($^{D}_{I}$)(R)(B)***
   (d) Conversione e Test: **TR(T)($^{D}_{I}$)(R)(B)**
8. Sincronizzazione Multi-Micro (uso speciale di Z ed S): **MREQ*, MBIT*, MRES*, MSET*.**

* Privilegiate

Figura 4.11 — Categorie Funzionali delle Istruzioni

del campo da -2^{n-1} a $2^{n-1}-1$ si ha il valore «*assoluto minore di*». Per esempio, potrebbero essere degli indirizzi di memoria. Quando viene eseguita la sottrazione x − y, C viene posto ad uno se e solo se *non* c'è riporto nell'addizione x + (−y). Perciò questa corrisponde al caso in cui se e solo se x < y, dove x ed y vengono considerati dei valori assoluti. (In realtà, se y = 0 non vi è riporto nell'addizione x + (−y), ma c'è un riporto nella condizione negata NEG(y), per cui nella sottrazione x − y, C non viene posto ad uno). In altre parole, x ULT y se e solo se C = 1 dopo l'istruzione CP x,y.

Nella Figura 4.12 sono riportati i 16 valori dei codici condizione, i loro significati e i loro mnemonici di assemblaggio.

Codice Condizione (esadecimale)	Significato	Mnemonico	Codice Condizione (esadecimale)	Significato	Mnemonico
0	Falso	Nessuno	8	Vero	Spazio*
1	S XOR V = 1	LT	9	S XOR V = 0	GE
2	LT OR Z = 1	LE	A	LT OR Z = 0	GT
3	C OR Z = 1	ULE	B	C OR Z = 0	UGT
4	V/P = 1	OV,PE	C	V/P = 0	NOV,PO
5	S = 1	MI	D	S = 0	PL
6	Z = 1	Z,EQ	E	Z = 0	NZ,NE
7	C = 1	C,ULT	F	C = 0	NC,UGE

* Caso di Default (esempio JR X ha 8 nel campo cc)

Figura 4.12 — Codici Condizione

Formato delle Istruzioni

Idealmente si dovrebbe interagire con il calcolatore ad un livello non più basso dei mnemonici dell'assemblatore. La codifica reale delle istruzioni in esadecimale dovrebbe essere irrilevante. Ma, in verità, ci sono molte ragioni per cui questo non si verifica:

- Istruzioni diverse occupano una diversa quantità di memoria. Qualche volta lo spazio di memoria ha un prezzo e allora è utile avere una esatta informazione dei formati.
- Di frequente gli strumenti di debug disponibili sono di tipo esadecimale (o anche peggio).
- A volte si ha bisogno di costruire una «pezza» — una sostituzione di una certa parte del programma assemblato. Può anche essere necessario scrivere in forma esadecimale la pezza senza dover ricorrere all'assemblatore.

- A volte può capitare di dover disassemblare manualmente un pezzo di programma esadecimale per il quale non è disponibile nessun tabulato del sorgente (source).

Inoltre, la conoscenza dei formati istruzione fornisce una buona base di partenza per la presentazione e la comprensione dei vari aspetti delle istruzioni.

Le istruzioni dello Z8000 seguono uno schema piuttosto semplice: sono tutte costituite da una o due parole di istruzione, seguite da una o due parole di indirizzo o argomento implicito. In generale, il primo byte della prima parola dell'istruzione determina il tipo di istruzione e di indirizzamento da essa usato. I byte rimanenti specificano gli argomenti e le variazioni.

Se i primi due byte dell'istruzione sono degli uni, allora si tratta di una delle quattro istruzioni speciali. Queste sono state previste per realizzare in una sola parola funzioni usate frequentemente, senza indirizzo addizionale o argomento implicito. Per queste quattro, la prima cifra esadecimale determina il tipo di istruzione, mentre i rimanenti dodici bit codificano la funzione desiderata. La Figura 4.14 riporta queste quattro istruzioni. La prima è un caricamento, realizzato con una sola parola, di un valore immediato di otto bit in un registro di un byte; le altre tre sono trasferimenti a locazioni indirizzate dal PC con piccoli spiazzamenti. Dal momento che tutte le istruzioni partono da un indirizzo di byte pari gli spiazzamenti sono interpretati come numero di parole: fino a 4096 per la CALR, 256 per JR e 128 per D(B)JNZ. Per CALR e JR gli spiazzamenti sono numeri con segno, che permettono di accedere ad una «fascia» centrata sull'istruzione. A D(B)JNZ corrisponde un numero assoluto che rappresenta un salto all'indietro per un massimo di 128 parole.

Figura 4.13 — Prima Approssimazione del Formato delle Istruzioni dello Z8000

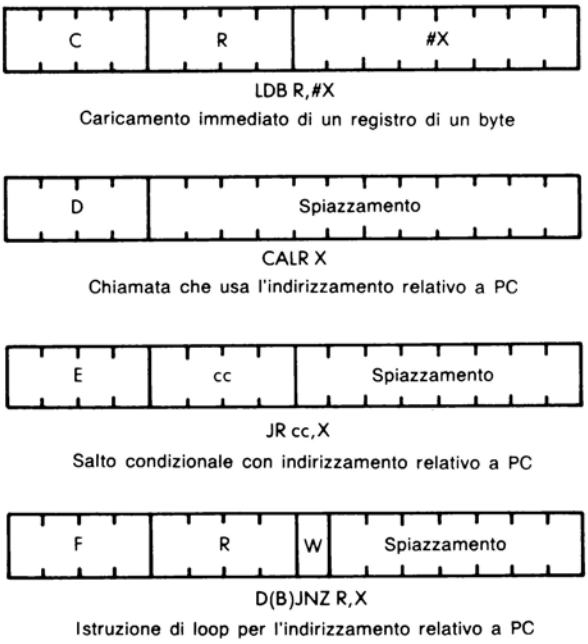


Figura 4.14 — Le Quattro Istruzioni Speciali Concepite per una Elevata Densità di Codifica

La Figura 4.14 illustra anche altre due convenzioni per la descrizione delle 45 istruzioni che saranno usate più avanti in questo capitolo: W e cc. Il simbolo W ha il valore uno o zero, a seconda che si consideri l'istruzione in versione a byte o a parola. In questo esempio, DJNZ avrà questo bit uguale ad uno, mentre DBJNZ lo avrà a zero. Il simbolo cc nel formato istruzione, rappresenta uno dei sedici possibili valori del campo a quattro bit del codice condizione. L'esempio delle istruzioni in linguaggio di assemblaggio, riportato sotto la figura del formato, rappresenta uno dei corrispondenti mnemonici di assemblaggio. (Vedasi Figura 4.12).

Se i primi due bit dell'istruzione non sono posti ad uno, questo campo è definito come campo di *modo*, e i sei bit adiacenti al primo byte sono denominati campo *codice operativo*. (Vedasi Figura 4.13). In generale, il byte successivo è diviso in due parti ciascuna indicante generalmente la locazione di uno degli argomenti. La Figura 4.15 riporta una istruzione con questa tipica struttura ed illustra anche le diverse convenzioni stabilite.

Per prima cosa si osservi l'istruzione in linguaggio di assemblaggio

EX(B) R,src

EX(B) R,src

W = 0 per EXB, quindi il codice operativo è 2C

W = 1 per EX, quindi il codice operativo è 2D

modo controlla il campo **src**; il campo **dst** è sempre un registro

Figura 4.15 — Una Tipica Istruzione a Due Operandi

riportata sotto la figura del formato. Questo significa che la figura rappresenta le istruzioni EX ed EXB, in funzione del valore assunto da W nella figura (2D è il codice operativo di EX, 2C di EXB). L'istruzione EX(B) possiede due argomenti: R ed src. Questa notazione significa che l'*argomento destinazione* (dst) di EX(B) deve essere un registro (parola o byte, in funzione del fatto che venga usata EX o EXB). *L'argomento sorgente* (src) può essere indirizzato con uno qualsiasi dei modi di indirizzamento; in particolare, quello che viene usato è specificato nel campo di modo. In generale, quando una istruzione richiede due argomenti il primo indicato nelle istruzioni in linguaggio di assemblaggio è chiamato *dst*, che generalmente indica dove va messo il risultato. Il secondo chiamato *src*, indica generalmente un argomento il cui valore non verrà alterato. Nella rappresentazione del formato, anteponiamo «dst» a R. Nel caso di EX(B) R,src questa è una ridondanza, ma per diciamo, ADC(B) R,R la rappresentazione del formato deve indicare di che tipo di R si tratta. Per coerenza, usiamo le parole «src» o «dst» nella rappresentazione del formato anche quando esiste una unica possibile interpretazione.

Nessuna istruzione dello Z8000 ha più di un campo per il modo. I modi di indirizzamento, tranne uno degli argomenti, sono impliciti nell'istruzione (esempio, il dst di una EX(B) deve essere un registro). In alcune istruzioni sono impliciti tutti i modi di indirizzamento per cui non c'è bisogno del campo di modo. In questo caso il campo di modo può essere usato per distinguere istruzioni diverse aventi lo stesso campo di codice operativo (esempio LDCTLB e RRDB). Più frequentemente, alcune istruzioni non potranno usare tutti e tre i possibili valori del modo. In questo caso la combinazione di quel codice operativo e della zona di modo utilizzata costituiscono la codifica di alcune altre istruzioni (esempio i codici operativi di RET e JP sono gli stessi; RET utilizza il valore di modo corrispondente al salto ad un registro).

Un'altra possibile variante è costituita dalle istruzioni ad un solo argomento. In questo caso il campo di modo può essere usato per determinare

il modo di indirizzamento per l'unico argomento, mentre il campo a quattro bit dell'argomento non utilizzato può essere usato per distinguere altri tipi di istruzioni con lo stesso codice operativo (esempio CLR(B), NEG(B), COM(B), TEST(B) hanno tutte lo stesso campo di codice operativo).

I campi a quattro bit vengono utilizzati per indicare argomenti siccome ogni modo di indirizzamento viene specificato in funzione di un registro. Ogni indirizzo o argomento immediato, implicito nel modo di indirizzamento, è appeso all'istruzione base. Questo utilizzo dei campi di quattro bit rende molto funzionali le istruzioni in esadecimale. Per esempio, (riferendosi alla Figura 4.15) il codice di EX R3,R2 è AD23 siccome il valore del campo di modo per un registro sorgente è 8. Questo illustra la nostra convenzione usata per esprimere i tre possibili valori del campo di modo. Siccome questi costituiscono la metà più significativa di una cifra esadecimale, li chiamiamo 0, 4, 8. In questo modo possono essere sommati alla cifra più significativa del campo del codice operativo (esempio il modo 8 più il codice operativo 2D è uguale ad AD). (Vedasi Figura 4.13).

Nel prossimo capitolo discuteremo in dettaglio i modi di indirizzamento dello Z8000. A questo punto abbiamo solamente bisogno di conoscere come vengono codificate, nei campi di modo, delle istruzioni. Per la maggior parte delle istruzioni esistono cinque modi principali di indirizzamento: registro (R), indiretto su registro (@), immediato (#), indicizzato (X) e indirizzo diretto (DA). I tre valori del campo di modo (0, 4, 8) non sono sufficienti a codificare i cinque modi di indirizzamento, per cui bisogna prendere a prestito qualche cosa dal campo a quattro bit del registro. Fortunatamente, uno dei modi non richiede di specificare il registro; perciò, devono essere sacrificate solamente due delle 48 combinazioni modo/registro usate dagli altri tre modi. La scelta è stata fatta in modo tale da restringere l'uso di R0 ai modi indicizzato e registro indiretto. La Figura 4.16 ne riporta la codifica. Come visto sopra, nell'istruzione, il campo di modo controlla solamente un modo di indirizzo.

Le altre sono fisse (o controllate tramite altri bit). La Figura 4.16 mostra che le codifiche sono leggermente differenti quando il campo controllato dal modo è un src oppure un dst. La differenza è dovuta al fatto che il modo immediato non ha bisogno dei campi dst, per cui R0 può essere usato in questo caso come un registro indirizzo. Questa distinzione dipende dall'uso *reale* dell'argomento. Per esempio (Vedasi Figura 4.15), il secondo argomento di EX(B) R,src viene chiamato src perchè appare come secondo nell'istruzione in linguaggio di assemblaggio, ma il modo di indirizzamento è codificato come un campo dst perchè i contenuti di R devono essere memorizzati in questo.

Campo di modo	Registro Src	Modo di Indirizzamento	Registro Dst	Modo di Indirizzamento
0	1-15	Indiretto su Registro(@)	0-15	Registro Indiretto (@)
0	0	Immediato (#)		
4	1-15	Indicizzato (X)	1-15	Indicizzato (X)
4	0	Indirizzo diretto (DA)	0	Indirizzo diretto (DA)
8	0-15	Registro (R)	0-15	Registro (R)

Figura 4.16 — Codifica del Modo di Indirizzamento Mediante Modo e Registro

Descrizione delle istruzioni

La descrizionie delle 45 istruzioni, alla fine di questo capitolo, fornisce delle informazioni dettagliate sul funzionamento, temporizzazione e formato di tutte le istruzioni dello Z8000. Ci sarebbero potuti essere più o meno di 45 gruppi; l'obiettivo era quello di raggruppare insieme istruzioni simili, ma anche di evitare allo stesso tempo le peripezie necessarie per rendere valida una spiegazione per due istruzioni superficialmente simili. Questo è il genere di considerazioni che si fa quando si deve decidere se allocare le funzioni di programma in moduli diversi o scrivere due blocchi similari di codici o estrarre le analogie per fare una subroutine. Spesso diviene solamente un problema di gusto. Sarà interessante guardare queste descrizioni scorrendole quando leggerete le seguenti spiegazioni dei formati.

Nell'angolo in alto a sinistra di ciascuna istruzione è riportato un numero compreso tra 1 e 45 usato per identificarle nell'indice descritto più avanti. Sempre in alto a sinistra è riportato il modello dell'istruzione in linguaggio di assemblaggio, usando la convenzione delle parentesi opzionali discusse prima. Sotto di questo, quando necessario, viene riportata, in una tabella di mnemonici in linguaggio di assemblaggio, una espansione delle opzioni ricavate da questa descrizione. Le opzioni (B) o ($^{B}_{L}$) non vengono sviluppate. Alcune descrizioni riguardano solamente un mnemonico (esempio Descrizione 15), mentre altre, principalmente istruzioni di blocco, riguardano molti mnemonici. (esempio la Descrizione 20 riguarda 16 mnemonici assembler, anche senza l'espansione della opzione (B)). L'*Indice Alfabetico delle Descrizioni delle Istruzioni* elenca 105 mnemonici di assemblaggio separati; per ciascun mnemonico viene fornito il numero della descrizione delle istruzioni ad esso relativo.

Sotto l'istruzione, espressa in linguaggio di assemblaggio, c'è l'elenco

mnemonico, ove viene dato il nome (o i nomi) dell'istruzione descritta. Sotto questa si trova la spiegazione del suo funzionamento. Nelle spiegazioni viene fatta una distinzione tra gli *argomenti* dell'istruzione (esempio R0) e i loro *valori* o contenuti (esempio −2758).

Nell'angolo in alto a destra della descrizione di ciascuna istruzione si trova un insieme di sei quadrati identificati con C, Z, S, V o P, D, H. Un punto nel quadrato sta ad indicare che quella istruzione agisce su quel determinato bit di FLAGS. Se non c'è un punto nel quadrato il corrispondente bit di FLAGS non viene toccato. Se i bit di FLAGS disegnati vengono usati secondo il loro significato principale (aritmetico) (Vedasi Figura 4.17) allora probabilmente nella descrizione non ci sarà nessun ulteriore riferimento a questi. Se esiste un qualche significato insolito connesso a questi bit, questo verrà evidenziato nella descrizione.

Sotto la descrizione viene riportata la Figura del formato. Questa è costituita da un rettangolo rappresentante i 16 bit dell'istruzione base (o qualche volta due rettangoli rappresentanti 32 bit). Il rettangolo è suddiviso in zone da barre verticali (si veda Figura 4.15). Ad eccezione del campo di *modo*, sebbene questi campi siano ulteriormente suddivisi, queste suddivisioni vengono quasi sempre effettuate sulle parti delimitanti delle cifre esadecimali. Se qualche istruzione ha sempre dei valori fissi in uno qualsiasi di questi campi, nel campo corrispondente viene scritta la sua rappresentazione esadecimale. Altrimenti vengono scritte alcune indicazioni tipo *op, src R, D/I, dst,* ecc.. Nel caso di *src* o *dst* senza l'indicazione di modo del tipo @ o R, il modo di indirizzamento corrispondente sarà controllato dal campo di modo. Una nota sotto la figura indicherà quali dei modi 0, 4, 8 è disponibile e sarà fornita esplicitamente una nota qualora questi modi non siano interpretati esattamente come specificato in Figura 4.16.

Se la descrizione dell'istruzione riguarda più di un codice operativo allora questo verrà indicato nel campo *op*, e sotto la figura del formato verranno elencati i vari valori possibili con i corrispondenti significati. L'indicazione del simbolo W nel codice operativo viene usata (vedasi Figura 4.15) per distinguere le versioni che operano con la parola o il byte. Un altro modo, per esprimere quanto appena detto, potrebbe essere il seguente: il bit 8 dell'istruzione viene posto ad 1 per operazioni sulla parola, azzerato nelle versioni per byte, ma siccome non tutte le istruzioni (anche all'interno della descrizione della stessa istruzione) usano il bit 8 in questo modo sembra più chiaro scrivere «op + W», dove W = 1 o W = 0, invece di rappresentare la suddivisione del campo come nella figura del formato.

In alcuni casi (esempio LDK R,#n) un argomento immediato viene codificato nell'istruzione e non sommato a questa, come una parola in più,

C: Riporto per l'addizione, «prestito» per la sottrazione che risulta fuori dall'estensione del campo src nella moltiplicazione e divisione. Bit di estensione del registro per gli spostamenti e le rotazioni.

Z: Risultato zero per operazioni e controlli dei dati; condizione verificata per il controllo di un blocco.

S: Risultato negativo (il bit più significativo ad 1) per operazioni e controlli dei dati: posizionamento di MI nella sincronizzazione multi-micro.

V: Superamento aritmetico (definito in vari modi) per operazioni aritmetiche. spostamenti/rotazioni e variazioni del contatore/puntatore; contatore a zero per le istruzioni di blocco.

P: Parità pari per le versioni a byte nelle operazioni logiche di controllo.

D: È stata realizzata una sottrazione di byte; viene azzerato quando si esegue una somma di byte.

H: Riporto dal digit meno significativo durante la somma o sottrazione di un byte.

Figura 4.17 — Condizioni Rappresentate dai Bit di FLAGS

come sarebbe stato se il «modo 0,R0» codificasse un campo *src*. In questo caso #n appare nel campo appropriato della figura del formato.

In alcuni casi (esempio LDCTL dst, src) un campo qualche volta è un src e qualche volta un dst. In questo caso viene dato un nome mnemonico ma trattato esattamente come src o dst. Qualche volta (esempio spostamenti e test di blocchi) le denominazioni src e dst sono prive di significato, perciò sono entrambe eliminate e viene esplicitamente indicato l'uso dei vari campi.

Alla destra della figura del formato è riportata la temporizzazione dell'istruzione. Per alcune istruzioni questa corrisponde ad un numero fisso di cicli. Più spesso consiste di una tabella che esprime i tempi in cicli, per i vari modi di indirizzamento o variazioni dell'istruzione. Qualche volta (esempio istruzione di blocco) la temporizzazione è espressa con una formula che è funzione di alcuni parametri tipo il numero di esecuzioni. Le temporizzazioni fornite sono riferite alla versione non-segmentata. Utilizzando la Figura 4.18 si possono ricavare facilmente da questi valori le temporizzazioni per la versione segmentata. La sola differenza si verifica quando un *indirizzo* viene appeso all'istruzione. Esiste la stessa differenza per le versioni byte, parola e parola lunga. I soli modi alterati sono l'indirizzamento diretto ed indicizzato. L'ammontare della differenza dipende dal fatto che vengono usati indirizzi segmentati corti o lunghi; oggetto di discussione nel Capitolo V. La Figura 4.18 deve essere utilizzata nel seguente modo: il numero di cicli in essa indicato va sommato al valore della temporizzazione non segmentata dato nella descrizione dell'istruzione a cui si fa riferimento. Se la temporizzazione fornita è una formula (esempio

11 + 14n) il numero viene sommato solamente alla parte costante e non al coefficiente di n.

	DA	X
Indirizzo corto	+1	+0
Indirizzo lungo	+3	+3

Figura 4.18 – Tempo (Cicli) Addizionale per l'Indirizzamento Segmentato.

Sotto la figura del formato sono riportate delle note speciali o degli avvertimenti, inoltre nella maggior parte dei casi viene fornito un esempio. Per primo, viene spiegato se l'esempio serve a dimostrare l'effetto delle operazioni dando le condizioni iniziali oppure se l'esempio vuole solamente illustrare una tecnica. Successivamente vengono date le istruzioni in linguaggio di assemblaggio. Se l'esempio serve per dimostrare l'effetto delle operazioni allora i commenti riportano i cambiamenti dei valori: dei registri della memoria e dei bit di FLAGS. Altrimenti i commenti illustrano lo scopo delle particolari istruzioni usate per implementare la tecnica rappresentata.

Indici

Ci sono due indici forniti insieme alle descrizioni dell'istruzione: uno alfabetico, tramite il mnemonico assembler, uno numerico, tramite il campo del codice operativo (bit 13-8). Per l'indice alfabetico è dato il numero della descrizione dell'istruzione corrispondente. Nell'indice numerico, un codice operativo può corrispondere a varie descrizioni. In questo caso viene dato il numero di ciascuna descrizione seguito dai mnemonici assembler corrispondenti al codice operativo che viene riportato nella relativa descrizione. In questo indice si sono riunite le coppie di codici operativi (esempio 00/01, 02/03) quando rappresentano le versioni byte e parola della stessa istruzione.

Come Utilizzare le Descrizioni Delle Istruzioni

In questo paragrafo riporteremo alcuni esempi dettagliati che illustrano l'uso delle descrizioni delle istruzioni.

Iniziamo guardando alla Descrizione di Istruzione n. 2; si noti che nell'angolo in alto a sinistra c'è

$$\binom{\text{ADD}}{\text{SUB}}\binom{\text{B}}{\text{L}}\ \text{R,src}$$

e sotto questo c'è

$$\text{ADD}\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right),\ \text{SUB}\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right)$$

La prima di queste linee ci dice, in forma compatta, quali istruzioni stiamo esattamente trattando in questa descrizione e quali forme esse assumono nel linguaggio assembler. In altre parole, tutte le istruzioni qui descritte hanno due argomenti: un registro (per la destinazione) e una sorgente (controllata dal campo di *modo*), per esempio:

```
ADDB   RH0,#5

SUBL   RR2,RR4

ADD    R3,@R1

SUB    R7,TABLE(R3)
```

La seconda delle due linee

$$\text{ADD}\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right),\ \text{SUB}\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right)$$

sviluppa la parte mnemonica dell'istruzione della prima linea. Può sembrare che qui non sia necessario, ma per il momento saltiamo alla Descrizione 20 per vedere i casi in cui la prima linea viene sviluppata in 16 mnemonici di istruzione. Avremmo potuto espandere ulteriormente la linea precedente in:

ADD, ADDB, ADDL, SUB, SUBB, SUBL

ma questo sembrava confondervi piuttosto che chiarirvi l'argomento. Nella Descrizione 20, questo avrebbe avuto come conseguenza 32 mnemonici assembler.

Sotto queste due prime linee c'è una terza linea; il nome dell'istruzione:

Addizione/Sottrazione

La parola «mnemonico» deriva dal greco *mnemnos* che significa memoria. Un buon nome mnemonico di una istruzione vi aiuta a ricordare quello che fa quella istruzione. ADD e SUB sono dei buoni mnemonici, per cui il nome dell'istruzione

Addizione/Sottrazione

sembra superfluo. Ma guardate alla Descrizione 44 dove i mnemonici sono

TRDB, TRDRB, TRIB, TRIRB, TRTDB, TRTDRB, TRTIB, TRTIRB

Qui è importante sapere che questa è l'istruzione di Trasferimento di un Blocco.

Sotto il nome, c'è una descrizione verbale di ciò che fa l'istruzione di Addizione/Sottrazione. Guardate indietro alla Figura 4.1. Per ADD, il valore sorgente e destinazione vengono posti nell'unità logico aritmetica, dove vengono sommati insieme ed il risultato memorizzato nel registro destinazione. Questi sono i meccanismi dell'istruzione. Il *risultato* è che il valore sorgente è sommato al valore contenuto nel registro destinazione ed il risultato è posto nel registro destinazione. La descrizione verbale ci illustra il risultato non necessariamente i meccanismi. Generalmente il programmatore non ha bisogno di trattare con i meccanismi di esecuzione dell'istruzione.

Alla descrizione verbale segue la figura del formato istruzione. Si noti che nell'istruzione Addizione/Sottrazione ci sono quattro campi:

modo, op, src, dst R

Il primo di questi è il campo di modo. Esso controlla il campo src. Questo significa che i campi di modo ed src vengono presi ed interpretati insieme coerentemente con la semiparte sinistra della tabella di Figura 4.16. Dovremo ritornare su questo argomento, ma ricordatevi che non avete bisogno di sapere nulla di questo se state giusto scrivendo le istruzioni in linguaggio di assemblaggio. In altre parole, quando scrivete

ADD R0, @ R4

il programma assemblatore si preoccupa di predisporre il campo di modo a zero e quello src a quattro. Avete bisogno di conoscere come vengono interpretati i campi di modo e src solamente quando cercate di determinare da soli la versione esadecimale dell'istruzione, senza ricorrere all'assemblatore, o quando state eseguendo un confronto con un codice esadecimale e volete determinare l'istruzione che esso rappresenta.

Naturalmente, quello che abbiamo appena detto si applica in ugual modo ai campi op e dst R. Quando scrivete

ADD R0, @ R4

l'assemblatore pone il campo op ad 1 e il dst R a 0: voi non avete bisogno di preoccuparvi di tutto questo.

Si osservi che ci sono due note sotto la figura del formato: una riguarda il campo op e l'altra riguarda i campi di modo ed src.
La notazione

modi src: 0, 4, 8

è una abbreviazione per dire che i campi src e modo devono essere interpretati esattamente come nella tabella di Figura 4.16. In particolare, questo significa che voi non potete scrivere

ADD R5, @ R0

ADD R5,TABLE(R0)

siccome la prima sarebbe interpretata come segnalazione di un argomento immediato e la seconda come un indirizzamento indiretto. Naturalmente, un adeguato assemblatore Z8000 rifiuterà queste istruzioni. Solamente quando state facendo da soli il lavoro dell'assemblatore avete bisogno di trattare le istruzioni al livello dei dettagli della Figura 4.16.

La notazione riguardante il campo op ci dice che questo può assumere sei diversi valori esadecimali: 0 = ADDB; 1 = ADD; 16 = ADDL; 2 = SUBB; 3 = SUB; 12 = SUBL. Ricordatevi che il simbolo W ha valore zero per una istruzione riferita al byte e uno per una istruzione riferita alla parola. Questa è semplicemente un'altra forma di abbreviazione.

Guardando alla destra della figura del formato troviamo una tabella dei tempi delle istruzioni. Lungo il lato superiore della tabella sono riportati i cinque possibili modi sorgente: R, #, @ , DA e X. Queste sono le abbreviazioni per i modi di indirizzamento *a registro, immediato, indiretto su registro, diretto e indicizzato* che saranno discussi nel Capitolo V. Generalmente, tutti questi modi richiedono un numero diverso di cicli di esecuzione in funzione delle diverse modalità con cui ottengono i loro argomenti (in realtà, # e @ sono sempre uguali). Sfortunatamente non c'è una formula che ci permetta di calcolarli tutti partendo da un qualche tempo base dell'istruzione, siccome le istruzioni variano in funzione della sovrapposizione dell'azione di calcolo e prelievo dell'operando. Inoltre sulla parte sinistra della tabella appaiono le indicazioni: «ADD(B), SUB(B)» e «ADDL, SUBL». Questo significa che la prima riga della tabella ci dà le temporizzazioni per ADD, ADDB, SUB, SUBB, la seconda riga per

ADDL e SUBL. La versione con parola lunga richiede più tempo perchè ci sono due operazioni da eseguire una dopo l'altra e c'è il doppio di memoria da trattare per eseguire il prelievo dei dati (nei modi non basati su registro).

Tutti questi tempi sono espressi in cicli. Il tempo reale richiesto dipende dalla frequenza del clock. A 4 MHz, 4 cicli = 1 microsecondo; perciò

ADD R0,R1

viene eseguita in un microsecondo, mentre

ADDL RR0,TABLE(R2)

viene eseguita in 4 microsecondi (16 cicli).

Occorre notare che tutti questi tempi sono riferiti al funzionamento non segmentato. I valori per il funzionamento in modo segmentato possono essere calcolati partendo da questi ed usando la tabella di Figura 4.18. Innanzitutto vengono toccati solo i modi DA e X. Come vedremo nel Capitolo V questi sono i modi di indirizzamento in cui l'indirizzo appare come parte dell'istruzione. Questi indirizzi sono di lunghezza pari ad una parola nel funzionamento non segmentato ed una o due parole nel funzionamento segmentato. Sembrerebbe perciò, che le istruzioni nella versione segmentata richiedano un ulteriore tempo di tre cicli (il tempo per un prelievo da memoria) quando sono usati gli indirizzi a due parole ed esattamente lo stesso numero di cicli, come nel funzionamento non-segmentato, quando sono usati indirizzi costituiti da una sola parola. Tutto quanto detto corrisponde quasi a verità. La sola eccezione è l'indirizzamento diretto che richiede un ciclo in più nell'indirizzo ad una parola nel funzionamento segmentato in confronto all'analogo nel funzionamento non segmentato. Per esempio, se vogliamo conoscere quanto dura un ADDL usando un indirizzo a due parole per il modo DA nel funzionamento segmentato guardiamo alla tabella di Figura 4.18 e troviamo + 3, poi sommiamo questo numero ai 15 cicli riportati nella Descrizione 2 ottenendo in tal modo 18 cicli. Analogamente, per un ADD o ADDB con lo stesso indirizzo di segmento, il tempo richiesto sarebbe di 12 cicli. Se in questi due esempi si fosse usato un indirizzo segmentato ad una parola, il tempo calcolato sarebbe stato di 16 e 10 cicli rispettivamente.

L'ultima cosa ancora da vedere nella Descrizione 2, prima di guardare l'esempio riportato in fondo, è l'insieme di quadrati in alto nell'angolo destro indicati con C, Z, S, V, D, H. Per ciascuna descrizione di istruzione appare nella stessa locazione un insieme di questo tipo, ma qualche volta V è sostituito da P o da V/P (o P/V). Questi sono i sei bit di FLAGS. Si

noti che dentro ciascun quadretto c'è un punto e che nella descrizione non appare nessun altro riferimento ai bit di FLAGS. Questo significa che tutti i bit sono gestiti da queste istruzioni come descritto in Figura 4.17. Riguardando la Figura 4.17 si nota che questo significa:

C: posto ad uno per indicare il riporto per l'addizione o prestito per la sottrazione. Azzerato se non c'è nulla di quanto sopra.

Z: posto ad uno se il risultato è zero, azzerato se non è zero.

S: posto ad uno se il risultato è negativo, azzerato se è positivo.

V: posto ad uno se c'è un supero (vedasi Capitolo I) azzerato se non c'è niente.

D: posto ad uno da SUBB. Azzerato da ADDB. Lasciato inalterato da ADD, SUB, ADDL, SUBL.

H: posto ad uno se c'è riporto dai quattro bit del digit meno significativo ai quattro bit del digit più significativo durante ADDB, SUBB. Azzerato se non c'è questo riporto durante ADDB, SUBB. Lasciato inalterato da ADD, SUB, SUBL, ADDL.

Questi quadretti costituiscono un modo coinciso per rappresentare una grossa quantità di informazioni.

Per ultimo, in fondo alla Descrizione 2, viene riportato un esempio sull'uso dell'istruzione Addizione/Sottrazione. Inizia fissando alcune condizioni iniziali: R0 contiene il valore 12, R1 contiene 15, i registri a byte RH2 e RL2 contengono 126 e 3. Poi ci sono due istruzioni in linguaggio assembler. I commenti riportano i valori dei registri coinvolti e i bit di FLAGS interessati *dopo* che è stata eseguita l'istruzione.

Per esempio, con le condizioni date prima,

ADDB RH2,RL2

esegue la somma di 3 contenuto in RL2, con 126 in RH2; il risultato è 129 e viene posto in RH2. Ma 129 è il complemento a due di −127, espresso in 8 bit, per cui si è avuto un superamento. I bit di FLAGS riflettono questo risultato: C = 0 siccome non si è avuto riporto (la somma avrebbe dovuto superare 255); Z = 0, siccome la somma non è zero; S = 1 siccome −127 è negativo; V = 1 siccome la somma dei due numeri positivi era negativa; D = 0 siccome l'istruzione era ADDB; H = 1 siccome si è avuto riporto nel sommare il digit meno significativo (14 + 3 = 17 = 16 + 1). Il 16 è il riporto al digit più significativo.

La prossima istruzione:

SUB R0, R1

non è una istruzione che opera sul byte; D e H non vengono toccati. Il risultato della sottrazione di 15 da 12 è -3. Perciò, C = 1 (si è verificato un prestito); Z = 0 siccome il risultato non era zero; S = 1 siccome -3 è negativo; V = 0 siccome non c'è stato superamento.

Nell'esempio riportato nella Descrizione 2 le due istruzioni sono indipendenti una dall'altra. Esse sono state date per illustrare cose diverse. D'altra parte, nella Descrizione 1 ci sono due sequenze di istruzioni. In ciascuna di queste sequenze, i valori del registro e dei bit di FLAGS risultanti da ciascuna istruzione, diventano i valori iniziali per la successiva istruzione. Questa dipendenza di ciascuna istruzione da quella che la precede è sottolineata dall'unione dei quadretti che contengono i valori di FLAGS.

Ora che abbiamo completamente analizzato una Descrizione di Istruzione, cercheremo di usarla per poter ottenere i codici esadecimali delle istruzioni nel caso di un programma reale – la subroutine LENGHT illustrata in Figura 3.7. Per fare questo useremo l'Indice Alfabetico delle Descrizioni delle Istruzioni per poter trovare le istruzioni che stiamo assemblando. Per fare un esempio concreto, assumeremo che il programma debba essere assemblato partendo dalla locazione di memoria 1000_{16}.

La prima riga è:

LENGTH: PUSH @ SR,R1

La parte LENGTH: determina l'assegnamento del valore 1000_{16} al simbolo LENGTH, ma siccome all'interno della subroutine non si fa mai riferimento a LENGTH, non abbiamo bisogno di usare questo particolare. Per assemblare l'istruzione, guardiamo il nostro indice e troviamo che PUSH viene riportato nella Descrizione Istruzione 36. Passiamo a questa descrizione, vi troviamo due figure di formato: una per le sorgenti immediate (esempio, PUSH @SR,#5) l'altra per tutti gli altri modi sorgente. La figura in alto è quella di cui abbiamo bisogno, siccome non abbiamo una sorgente immediata; infatti la nostra istruzione usava il modo registro (R).

Riempiamo i quattro campi muovendoci da sinistra a destra: modo = 8 per il modo registro (vedasi Figura 4.16); op = 13 per PUSH (si veda la notazione riportata sotto la figura); per dst @ R metteremo il numero del registro SR (questo sarà spiegato nel Capitolo V). Dal momento che il no-

stro esempio è scritto per il funzionamento non segmentato, il registro di stack è R15. La rappresentazione esadecimale di 15 è F, per cui poniamo il campo dst @ R = F. Infine per fissare R1 poniamo src = 1. Perciò,

PUSH @SR,R1 è uguale a 93F1

Si noti come il modo di 8 è stato sommato alla prima cifra del codice operativo, 13, onde ottenere 93 come valore esadecimale della prima metà della parola.

Analogamente l'istruzione successiva

PUSH @ SR,R2 è uguale a 93F2

L'unica differenza sta nel fatto che il campo src è uguale a 2 (per R2) invece che ad 1 (per R1).

Per l'istruzione successiva

CLR R0

l'indice ci dice di guardare alla Descrizione 6. Riempiamo i campi come prima, per indicare il modo registro abbiamo il modo = 8; op = 0D (0C + 1, siccome W = 1 per CLR,0 per CLRB); dst = 0 per R0. Il quarto campo è già stato riempito per noi; la cifra finale, in questo caso un 8, è quello che distingue CLR da NEG, COM e altre istruzioni. Mettendoli insieme, otteniamo

CLR R0 uguale a 8D08

Analogamente l'istruzione successiva

CLRB RL2 è uguale a 8CA8

Nella seconda cifra delle ultime due istruzioni si noti la differenza tra C e D. Questo si verifica perchè W = 0 per CLRB, per cui il codice operativo è 0C; e sommandogli il valore 8, per indicare il modo registro si ottiene 8C. L'altra cosa da notare è A per RL2. Tutti gli altri registri che abbiamo visto fino ad ora sono stati codificati per mezzo dei loro numeri: R0 è stato codificato con zero, R1 con uno, R2 con due. Questo è vero anche per i registri a parola lunga – RR2 è codificato con 2, ecc. Ma per i registri a byte non possiamo codificare sia RL2 che RH2 con 2. Abbiamo bisogno di un mezzo per distinguerli. Allora per i registri a byte RL, al numero corrispondente al registro viene sommato il valore 8: RL0 viene co-

dificato con 8, RL1 con 9, RL2 con A (come visto sopra), ecc. Tutto questo verrà descritto nel Capitolo V.

Poi arriviamo a

CPIRB RL2,@R1,R0,EQ

che l'indice ci dice essere illustrata nella Descrizione 10. Questa istruzione non ha un campo di modo, siccome la sola variazione permessa è dst = @ o dst = R come illustrato nella notazione per l'opzione S (stringa), sotto la figura del formato. Il campo op assume il valore BA, essendo W = 0 per la versione byte. Il nostro campo src @ R assume il valore 1 per R1. L'ultima cifra della prima parola ha D/I = 0 per l'incremento, R = 1 per la ripetizione ed S = 0 per dst = R. Questo da 0100_2, o 4_{16}; perciò la prima parola è uguale a BA14. Nella seconda parola il campo cnt R è 0 per R0, il campo dst è A per RL2 ed il campo cc è 6 per EQ (vedasi Figura 4.12). Questa da per la seconda parola 00A6; perciò

CPIRB RL2,@R1,R0,EQ è uguale a BA14/00A6

Ora dovreste essere in grado di codificare le cinque istruzioni rimanenti:

NEG R0	è uguale a 8D02
DEC R0	è uguale a AB00
POP R2,@SR	è uguale a 97F2
POP R1,@SR	è uguale a 97F1
RET	è uguale a 9E08

Il programma completamente assemblato è riportato in Figura 4.19. La colonna più a sinistra contiene gli indirizzi in esadecimale e la successiva il codice macchina corrispondente all'istruzione sorgente, che appare nella colonna più a destra. Si noti anche che nell'assemblare il programma non abbiamo mai usato indirizzi esadecimali: le stesse 11 parole di codice potrebbero essere piazzate ad un generico indirizzo, e continuerebbero a fare esattamente la stessa cosa. Questo è conosciuto come codice *non dipendente dalla posizione.*

In realtà nella versione della subroutine del programma di crittografia (Figura (3.9), la subroutine MOD23 (Figura 3.8) e la subroutine LENGTH che abbiamo appena assemblato hanno solo due istruzioni che dipendono dagli indirizzi reali a cui sono assemblati i programmi. Queste istruzioni, CALL LENGTH e CALL MOD23, sono entrambe illustrate in Figura 3.9. Esse usano il modo di indirizzamento diretto.

```
Indirizzo  Contenuto
  1000       93F1      LENGTH: PUSH   @SR,R1
  1002       93F2              PUSH   @SR,R2
  1004       8D08              CLR    R0
  1006       8CA8              CLRB   RL2
  1008       BA14              CPIRB  RL2,@R1,R0,EQ
  100A       00A6
  100C       8D02              NEG    R0
  100E       AB00              DEC    R0
  1010       97F2              POP    R2,@SR
  1012       97F1              POP    R1,@SR
  1014       9E08              RET
```

Figura 4.19 — Versione Assemblata di LENGTH

Dalla Descrizione Istruzione 5 possiamo vedere come assemblare queste istruzioni. Se la subroutine LENGTH è localizzata a 1000 (come in Figura 4.19), allora (usando modo = 4, dst = 0 — vedasi Figura 4.16),

CALL LENGTH è uguale a 5F00/1000

L'indirizzo reale di LENGTH è la seconda parola dell'istruzione.

Come esempio finale dell'uso delle Descrizioni Istruzioni, disassembleremo un programma. La Figura 4.20 riporta una parte di codice esadecimale che noi desideriamo tradurre in mnemonici assembler.

Iniziamo con ABF9 alla locazione 3000. Avendo A = 8 + 2 sarà un modo = 8, op = 2B. Guardiamo all'indice numerico e troviamo la riga

2A/2B 12 DEC(B)

Questo ci dice che 2B è un'istruzione di tipo DEC e che la Descrizione 12 ci darà il suo formato. Saltando alla Descrizione 12 troviamo che

ABF9 è uguale a DEC R15,#10

dal momento che modo = 8 significa modo registro (vedasi Figura 4.16) F sarà la codifica di R15 e 9 = n − 1.

L'istruzione successiva alla locazione 3002 è 1CF9. Il nostro indice numerico ci dice che 1C è un'istruzione LDM e di fare riferimento alla Descrizione 28. In questa Descrizione troviamo che la cifra finale 9 indica

che si tratta di un'istruzione LDM da registri a memoria. F è la codifica di R15, e siccome il modo vale 0, si ha @ R15. Inoltre per trovare che il registro è R0 e il numero di registri è 5 si vede che dobbiamo guardare alla prossima parola 0004 posta in 3004. Abbiamo perciò che

1CF9/0004 è uguale a LDM @ SR,R0,#5

Essendo 0004 parte dell'istruzione LDM, la prossima parola che dobbiamo considerare è A104 posta a 3006. Essendo A = 8 + 2 ne consegue modo = 8, op = 21. Guardando l'indice numerico sotto 21 veniamo indirizzati alla descrizione 23 per l'istruzione LD. 04 significa che src = R0, dst = R4; perciò

A104 è uguale a LD R4,R0

Riconosciamo nelle due parole successive una CALL alla subroutine localizzata a 1000 (o se non la riconosciamo, il nostro indice ci dice che op = 1F è una CALL). Dal momento che non sappiamo quale subroutine si trovi in 1000 in questo esempio possiamo chiamarla SUB1. Perciò,

5F00/1000 è uguale a CALL SUB1

Ci troviamo ora all'indirizzo 300C dove c'è A103; dovreste essere in grado di trovare da soli che:

A103 è uguale a LD R3,R0
8D28 è uguale a CLR R2
2048 è uguale a LDB RL0,@R4
8C84 è uguale a TESTB RL0

Questo ci porta ad E609 posto a 3014. E = C + 2, ma non esiste modo = C. Per cui questa è una delle quattro istruzioni speciali (vedasi Figura 4.14). E = JR; 6 codifica un codice condizione (Figura 4.12); e 09 è lo spiazzamento. JR è spiegato nella Descrizione 22 dove troviamo come dobbiamo interpretare lo spiazzamento. Prendiamo 09, lo moltiplichiamo per 2 ed otteniamo 12 (ricordatevi che questo è in esadecimale) e lo sommiamo all'indirizzo dell'istruzione successiva (esempio 3016) per cui la destinazione di JR è 3016 + 12 = 3028. Tutto ciò che noi possiamo fare è fissare una label (etichetta) per 3028; la chiameremo XX. Infine, guardando la Figura 4.12, si vede che il codice condizione 6 è uguale a Z o EQ. Dal momento che l'istruzione precedente era TESTB RL0, la chiameremo Z (sebbene non faccia differenza). Per cui

E609 è uguale a JR Z,XX

La prossima istruzione, 7010, ci impegna un po' di più. Guardando il nostro indice sotto op = 30, troviamo Descrizione 11, DAB, Descrizione 17, EXTS($^{B}_{L}$) e Descrizione 23, LD(B). La DAB sembra la meno probabile, ma la guardiamo ugualmente, troviamo solamente B0, esempio, modo = 8 op = 30. Siccome abbiamo modo = 4, possiamo proseguire. Anche EXTS non sembra essere quella buona, e quando la analizziamo troviamo il modo = 8, op = 31. Perciò, LD(B) è quella che cerchiamo. In realtà, è l'ultimo formato mostrato nella Descrizione 23: un Caricamento Indicizzato tramite Base. Guardando al formato dovreste essere in grado di vedere che:

7010/0200 è uguale a LDB RH0,R1(R2)

Indirizzo	Contenuto
3000	ABF9
3002	1CF9
3004	0004
3006	A104
3008	5F00
300A	1000
300C	A103
300E	8D28
3010	2048
3012	8C84
3014	E609
3016	7010
3018	0200
301A	8808
301C	2E48
301E	A940
3020	A920
3022	5F00
3024	2000
3026	E8F4
3028	1CF1
302A	0004
302C	A9F9
302E	9E08

Figura 4.20 — Programma Misterioso

Da questo punto in poi dovreste essere in grado di finire l'esempio da soli. Che relazione ha questo programma con la Figura 3.9?

Alcune Considerazioni di Progetto

L'autore è ben conscio che esistono molti modi per presentare il tipo di informazioni illustrate nelle Descrizioni Istruzioni. L'autore è anche consapevole che questa presentazione non è conforme allo stile largamente usato nei manuali tecnici. Nel seguito sono riportate alcune domande che l'autore si è posto prima di scrivere questo capitolo. Le sue risposte sono qui in questo libro; lui accoglierà positivamente le opinioni degli altri su questo soggetto, specialmente da quelli che hanno avuto la possibilità di usare questo libro come aiuto al proprio lavoro con lo Z8000.

- La Descrizione Istruzioni dovrebbe essere un efficiente strumento (come l'oscilloscopio) da usare, ma che non può essere totalmente utilizzato se privi di ogni preparazione?
- Gli obiettivi principali come la codifica del modo (Figura 4.16) i codici condizione (Figura 4.12) e la interpretazione di FLAGS (Figura 4.17) dovrebbero apparire solo in un luogo ed essere possibilmente ricordato dal programmatore, o dovrebbero essere ripetuti in ogni luogo ove vengono usati?
- Dovrebbe esserci una descrizione delle istruzioni per ciascuno dei 105 mnemonici istruzione identificabili separatamente (187 se si considerano separatamente le opzioni B ed L), o si dovrebbero evidenziare le analogie dei gruppi di istruzioni trattandoli insieme?
- Si dovrebbe usare la rappresentazione esadecimale, sebbene porti a dei paradossi come i modi di 0,4 e 8, o quella binaria?

Queste sono alcune delle domande che l'autore si è posto esplicitamente. Ci sono alcune altre scelte che l'autore può aver fatto implicitamente trascurando altre possibili alternative. Infine ricordiamo ancora che ogni commento da parte del lettore sarà sempre ben accettato.

INDICE DELLE DESCRIZIONI DELLE ISTRUZIONI
(Alfabetico)

ADC(B)	1	INC(B)	12	OTDR(B)	20
ADD($^{B}_{L}$)	2	IND(B)	20	OTIR(B)	20
AND(B)	3	INDR(B)	20	OUT(B)	19
BIT(B)	4	INI(B)	20	OUTD(B)	20
CALL	5	INIR(B)	20	OUTI(B)	20
CALR	5	IRET	21	POP(L)	35
CLR(B)	6	JP	22	PUSH(L)	36
COM(B)	7	JR	22	RES(B)	4
COMFLG	8	LD($^{B}_{L}$)	23	RESFLG	8
CP($^{B}_{L}$)	9	LDA	24	RET	37
CPD(B)	10	LDAR	24	RL(B)	38
CPDR(B)	10	LDCTL(B)	25	RLC(B)	38
CPI(B)	10	LDD(B)	26	RLDB	39
CPIR(B)	10	LDDR(B)	26	RR(B)	38
CPSD(B)	10	LDI(B)	26	RRC(B)	38
CPSDR(B)	10	LDIR(B)	26	RRDB	39
CPSI(B)	10	LDK	27	SBC(B)	1
CPSIR(B)	10	LDM	28	SC	40
DAB	11	LDPS	29	SDA($^{B}_{L}$)	41
DEC(B)	12	LDR($^{B}_{L}$)	30	SDL($^{B}_{L}$)	41
DI	13	MBIT	31	SET(B)	4
DIV	14	MREQ	32	SETFLG	8
D(B)JNZ	15	MRES	31	SIN(B)	19
EI	13	MSET	31	SIND(B)	20
EX(B)	16	MULT(L)	33	SINDR(B)	20
EXTS($^{B}_{L}$)	17	NEG(B)	7	SINI(B)	20
HALT	18	NOP	34	SINIR(B)	20
IN(B)	19	OR(B)	3	SLA($^{B}_{L}$)	41

SLL($^{B}_{L}$)	**41**	SRL($^{B}_{L}$)	**41**	TRIRB	**44**
SOTDR(B)	**20**	SUB($^{B}_{L}$)	**2**	TRTDB	**44**
SOTIR(B)	**20**	TCC(B)	**42**	TRTDRB	**44**
SOUT(B)	**19**	TEST($^{B}_{L}$)	**43**	TRTIB	**44**
SOUTD(B)	**20**	TRDB	**44**	TRTIRB	**44**
SOUTI(B)	**20**	TRDRB	**44**	TSET(B)	**45**
SRA($^{B}_{L}$)	**41**	TRIB	**44**	XOR(B)	**3**

INDICE DELLE DESCRIZIONI DELLE ISTRUZIONI
(Mediante il valore numerico del codice operativo)

OP	Descrizione(i)		OP	Descrizione(i)	
00/01	2	ADD(B)	18/19	33	MULT(L)
02/03	2	SUB(B)	1A/1B	14	DIV(L)
04/05	3	OR(B)	1C	28	LDM
06/07	3	AND(B)		43	TESTL
08/09	3	XOR(B)	1D	23	LDL
0A/0B	9	CP(B)	1E	22	JP
0C/0D	6	CLR(B)		37	RET
	7	NEG(B),COM(B)	1F	5	CALL
	8	$\left(\begin{smallmatrix}\text{COM}\\ \text{RES}\\ \text{SET}\end{smallmatrix}\right)$FLG	20/21	23	LD(B)
	9	CP(B)	22/23	4	RES(B)
	23	LD(B)	24/25	4	SET(B)
	25	LDCTLB	26/27	4	BIT(B)
	34	NOP	28/29	12	INC(B)
	36	PUSH	2A/2B	12	DEC(B)
	43	TEST(B)	2C/2D	16	EX(B)
	45	TSET(B)	2E/2F	23	LD(B)
0E/0F		Non utilizzato		42	TCC(B)
10	9	CPL	30/31	11	DAB
11	36	PUSHL		17	EXTS$\left(\begin{smallmatrix}\text{B}\\ \text{L}\end{smallmatrix}\right)$
12	2	SUBL		23	LD(B)
13	36	PUSH		30	LDR(B)
14	23	LDL	32/33	23	LD(B)
15	35	POPL		30	LDR(B)
16	2	ADDL		38	R$\left(\begin{smallmatrix}\text{L}\\ \text{R}\end{smallmatrix}\right)$(C)(B)
17	35	POP		41	S$\left(\begin{smallmatrix}\text{D}\\ \text{L}\\ \text{R}\end{smallmatrix}\right)\left(\begin{smallmatrix}\text{L}\\ \text{A}\end{smallmatrix}\right)\left(\begin{smallmatrix}\text{B}\\ \text{L}\end{smallmatrix}\right)$

OP		Descrizione(i)	OP		Descrizione(i)
34/35	**1**	ADC(B)		**21**	IRET
	23	LDL		**26**	LD($^{D}_{I}$)(R)(B)
	24	LDA,LDAR		**31**	M($^{BIT}_{RES}$ $^{}_{SET}$)
	30	LDRL		**32**	MREQ
36/37	**1**	SBC(B)	3C/3D	**13**	($^{D}_{E}$)I
	23	LDL		**19**	IN(B)
	24	LDA		**25**	LDCTL
38	**44**	TR(T)($^{D}_{I}$)(R)B		**27**	LDK
39	**29**	LDPS		**39**	RRDB
3A/3B	**10**	CP(S)($^{D}_{I}$)(R)(B)	3E/3F	**19**	OUT(B)
	18	HALT		**39**	RLDB
	19	(S)($^{IN}_{OUT}$)(B)		**40**	SC
	20	(S)($^{IN}_{OUT}$)($^{D}_{I}$)(R)(B)			

Le quattro Istruzioni Speciali

Primo digit	Descrizione	
C	**23**	LD(B)
D	**5**	CALR
E	**22**	JR
F	**15**	D(B)JNZ

1. $\binom{AD}{SB}$C(B) dst R,src R

ADC(B), SBC(B)

C	Z	S	V	D	H
•	•	•	•	•	•

Addizione/Sottrazione con riporto

La somma del valore sorgente con il bit C è sommata al/sottratta dal valore destinazione; il risultato sostituisce il valore destinazione.

op	src R	dst R

Durata (cicli): 5

op: $\begin{cases} \mathbf{B4 + W = ADC(B)} \\ \mathbf{B6 + W = SBC(B)} \end{cases}$

Esempio: RH0 = 0; RL0 = −1; RH1 = 2; RH4 = 0; RL4 = 1; RH5 = 3.

ADDB RH1,RH5	**!RH1 = 5; RH5 = 3;**	0	0	0	0	0	0	!
ADCB RL0,RL4	**!RL0 = 0; RL4 = 1;**	1	1	0	0	0	1	!
ADCB RH0,RH4	**!RH0 = 1; RH4 = 0;**	0	0	0	0	0	0	!
		C	Z	S	V	D	H	

Così, il numero a 24-bit, 259 immagazzinato in RH4,RL4,RH5 è stato addizionato all'accumulatore a 24-bit RH0,RL0,RH1. Il valore iniziale dell'accumulatore era 65.282; il valore finale è 65.541 ed il valore finale di C = 0 indica che la somma non ha prodotto riporto.

Con gli stessi valori iniziali

SUBB RH1,RH5	**!RH1 = −1; RH5 = 3;**	1	0	1	0	1	1	!
SBCB RL0,RL4	**!RL0 = −3; RL4 = 1;**	0	0	1	0	1	0	!
SBCB RH0,RH4	**!RH0 = 0; RH4 = 0;**	0	1	0	0	1	0	!
		C	Z	S	V	D	H	

In questo caso, 65.282 − 259 = 65.023, e lo stato finale di C = 0 indica che la differenza non ha richiesto un prestito.

2. $\binom{\text{ADD}}{\text{SUB}}\binom{\text{B}}{\text{L}}$ R,src

ADD($\binom{B}{L}$), SUB($\binom{B}{L}$)

C	Z	S	V	D	H
•	•	•	•	•	•

Addizione/Sottrazione

Il valore sorgente è sommato al/sottratto dal valore destinazione ed il risultato sostituisce il valore destinazione.

modo	op	src	dst R

modi src: 0, 4, 8

	src R	src #	src @	src DA	src X
ADD(B),SUB(B)	4	7	7	9	10
ADDL,SUBL	8	14	14	15	16

Durata (cicli)

op:
- **00 + W: ADD(B)**
- **16: ADDL**
- **02 + W: SUB(B)**
- **12: SUBL**

Esempio: **R0 = 12; R1 = 15; RH2 = 126; RL2 = 3.**

ADDB RH2,RL2 = −127; RL2 = 3;

C	Z	S	V	D	H
0	0	1	1	0	1

!

SUB R0,R1 = −3; R1 = 15;

C	Z	S	V
1	0	1	0

!

3. $\left(\begin{matrix}\text{AND}\\ \text{OR}\\ \text{XOR}\end{matrix}\right)$(B) R,src

AND(B), OR(B), XOR(B)

C	Z	S	P	D	H
	•	•	•		

And/Or/Or Esclusivo

Il risultato delle operazioni logiche indicate sui valori sorgente e destinazione sostituisce il valore destinazione.

modo	op	src	dst R

modi src: 0, 4, 8

src

R	#	@	DA	X
4	7	7	9	10

Durata (cicli)

op: $\begin{cases} 06 + W: & \text{AND(B)} \\ 04 + W: & \text{OR(B)} \\ 08 + W: & \text{XOR(B)} \end{cases}$

Esempio: RH0 = $7A_{16}$; RL0 = 64_{16}; R1 = $F070_{16}$; R2 = $10FF_{16}$.

```
XORB RH0,RL0   !RH0 = 1E16;     RL0 = 6416;   [0|0|1]  !
                                              Z S P

AND  R1,R2     !R1 = 107016;    R2 = 10FF16;  [0|0]    !
OR   R1,R2     !R1 = 10FF16;    R2 = 10FF16;  [0|0]    !
                                              Z S
```

4. $\left(\begin{matrix}\text{BIT}\\ \text{RES}\\ \text{SET}\end{matrix}\right)$(B) dst, src

BIT(B), RES(B), SET(B)

	•				
C	Z	S	V	D	H

(solamente BIT(B))

Test/Azzeramento/Posizionamento ad uno di un bit

Il bit in questione viene posto ad uno per SET(B) azzerato per RES(B) e controllato (e lasciato invariato) per BIT(B). Con BIT(B), Z è posto a 1 se il bit è 0, azzerato se il bit è 1. SET(B) e RES(B) non influenzano i bit di stato.

Il bit in questione è quel bit del valore destinazione il cui numero è nei tre o quattro bit meno significativi del valore sorgente. I bit sono numerati da 0, per il bit meno significativo (quello più a destra) fino a 7 o 15 per quello più significativo (quello più a sinistra).

modo	op	dst	#Bit no.

modi dst: 0, 4, 8

SORGENTE IMMEDIATA

	dst R	dst @	dst DA	dst X
BIT(B)	4	8	10	11
RES(B)/SET(B)	4	11	13	14

Durata (cicli)

op		0	src R
0	dst R	0	0

SORGENTE DINAMICA

Durata (cicli): 10

op: $\begin{cases} \textbf{26 + W: BIT(B)} \\ \textbf{22 + W: RES(B)} \\ \textbf{24 + W: SET(B)} \end{cases}$

(per entrambi i formati)

Nota: Il registro *src* per una sorgente dinamica è un registro a *parola* sia per la versione a byte che per quella a parola. Per la versione a byte può variare solo R0 fino a R7, per le versioni a parola possono essere usati da R0 a R15.

Esempio: R0 = 1; R2 = 8; R3 = 0; R4 = $FFFF_{16}$.

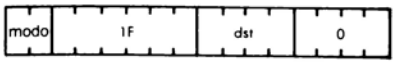
```
SET   R0,R2     !R0 = 101₁₆; R2 = 8          !
SETB  RL0,R4    !R0 = 181₁₆; R4 = FFFF       !
RES   R0,R3     !R0 = 180₁₆; R3 = 0          !
BITB  RH0,#1    !R0 = 180₁₆; Z = 1           !
BITB  RH0,R3    !R0 = 180₁₆; R3 = 0; Z = 0   !
```

5. $\binom{\text{CALL}}{\text{CALR}}$ dst

C	Z	S	V	D	H

Chiamata di Subroutine

Il PC viene salvato tramite il registro di stack (R15 o RR4); quindi l'indirizzo *destinazione* (NB. non è il *valore* destinazione come nelle altre istruzioni) sostituisce il contenuto del PC.

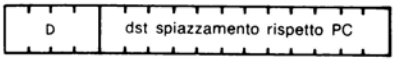


modi dst: 0, 4, (non 8)

@	DA	X
10	12	13

Durata (cicli)

CALL



Durata (cicli): 10

CALR

Per CALR, l'indirizzo destinazione è calcolato come segue: lo spiazzamento viene preso come un valore segnato di 12-bit, moltiplicato per 2, quindi *sottratto dalla* (NB. non *sommato* alla) locazione che segue la CALR. (Poiché CALR può essere usata come indirizzo destinazione nell'intervallo **$** − 4092 a **$** + 4098).

Note: (1) CALR non cambierà mai la parte del *segmento* di un indirizzo.

(2) La durata di CALL e CALR nel modo segmentato è superiore di 5 cicli rispetto ai valori calcolati usando la Figura 4.18

Durata (cicli): 15

CALR

	@	DA	X
indirizzo corto		18	18
indirizzo lungo	15	20	21

Durata (cicli)

CALL

Esempio: RR14 = (300_{16}, 100_{16}); PC = (200_{16}, 10_{16}); l'indirizzo di XYZ è nel segmento 1, spiazzamento 200_{16}; locazioni della memoria di stack nel segmento 3: **FA_{16} = 0000; FC_{16} = 0000; FE_{16} = 0000; 100_{16} = 1000_{16}.**

```
CALL  XYZ !RR14 = (300₁₆,FC₁₆); PC = (100₁₆, 200₁₆);
          Locazioni della memoria di stack nel segmento 3:
          FA₁₆ = 0000; FC₁₆ = 200₁₆; FE₁₆ = 10₁₆;
          100₁₆ = 1000₁₆!
```

In questo esempio, l'indirizzo della locazione che segue la CALL è stato salvato nello stack controllato da RR14 ed il nuovo valore di PC è l'indirizzo XYZ.

6. CLR(B) dst

C	Z	S	V	D	H

(Nota: Z non è posto ad uno)

Azzeramento

Al valore destinazione viene sostituito il valore zero.

modo	0C + W	dst	8

modi dst: 0, 4, 8

dst

R	@	DA	X
7	8	11	12

Durata (cicli)

Esempio: **R1 = 1000_{16}; R2 = 1002_{16}; memoria dati: 1000_{16} = −1; 1002_{16} = −1.**

CLR @R1 !R1 = 1000_{16}; locazione della memoria 1000_{16} = 0!

CLRB @R2 !R2 = 1002_{16}; locazione della memoria 1002_{16} = FF_{16}!

Nota speciale: Per le versioni future dello Z8000 si sta considerando l'istruzione CLRL con la forma seguente.

modo	1C	dst	0

7. $\binom{\text{COM}}{\text{NEG}}$(B) dst

COM(B), NEG(B)

C	Z	S	P/V	D	H
•	•	•	•		

Complemento/Negazione

Il valore destinazione viene sostituito dal suo complemento/negativo. Z, S e P sono posizionati concordemente con il risultato di COM(B). Per NEG(B), Z ed S sono come previsto, C è posizionato ad uno se il risultato non è zero e V è posto ad uno se il risultato è -2^{15} per NEG e -2^{7} per NEGB.

modo	0C + W	dst	op2

modi dst: 0, 4, 8

R	@	DA	X
7	12	15	16

Durata (cicli)

op2: $\begin{cases} \textbf{0: COM(B)} \\ \textbf{2: NEG(B)} \end{cases}$

Esempio: **R0 = −1; R1 = −1; RH2 = −128; RL2 = F_{16}.**

NEG R0	**!R0 = 1;**	C: 1, Z: 0, S: 0, V: 0	!
COM R1	**!R1 = 0;**	Z: 1, S: 0	!
NEGB RH2	**!RH2 = −128;**	C: 1, Z: 0, S: 1, V: 1	!
COMB RL2	**!RL2 = $F0_{16}$;**	Z: 0, S: 1, P: 1	!

8. (COM / RES / SET)FLG nome del bit di flag

COMFLG, RESFLG, SETFLG

C	Z	S	P/V	D	H
•	•	•	•		•

Complemento/Azzeramento/Posizionamento dei Bit di FLAGS

Ognuno dei bit di FLAGS designato nell'istruzione è complementato/azzerato/posizionato in FLAGS. COMFLAG lascia inalterato H.

8D	C	Z	S	P/V	op2

Durata (cicli): 7

op2:
- **5: COMFLG**
- **3: RESFLG**
- **1: SETFLG**

Esempio:

			C	Z	S	P/V	
RESFLG	**P,Z,C,S**	!	0	0	0	0	!
COMFLG	**P,S**	!	0	0	1	1	!
SETFLG	**C,V**	!	1	0	1	1	!
COMFLG	**C,Z,P**	!	0	1	1	0	!
RESFLG	**V,S**	!	0	1	0	0	!

9. CP($^{B}_{L}$) dst,src

C	Z	S	V	D	H
•	•	•	•		

Confronto

I bit di FLAGS sono posizionati come essi dovrebbero essere quando viene eseguita la sottrazione del valore sorgente dal valore destinazione.

modo	op	src	dst P

modi src: 0, 4, 8

op: $\begin{cases} 0A + W: & CP(B) \\ 10: & CPL \end{cases}$

CP($^{B}_{L}$) (dst = R)

	src R	#	@	DA	X
CP(B)	4	7	7	9	10
CPL	8	14	14	15	16

Durata (cicli)

modo	0C + W	dst	1

modi dst: 0, 4 (non 8)

CP(B)(dst ≠ R, src = #)

dst @	DA	X
11	14	15

Durata (cicli)

Esempio: **RR0 = (−1, −1); RR2 = (0,0); R4 = −32767 (= 8001_{16}).**

CPL RR0,RR2 ! 0 0 1 0 ! (C Z S V)

CPB RH0,RL1 ! 0 1 0 0 ! (C Z S V)

CP R4,#2 ! 0 0 0 1 (C Z S V) **(−32767 − 2 = 32767)!**

10. CP(S)$\binom{D}{I}$(R)(B) dst, src @R,cnt R,cc

CPD(B), CPDR(B), CPI(B) CPIR(B),
CPSD(B), CPSDR(B), CPSI(B), CPSIR(B)

•	•	•	•		
C	Z	S	V	D	H

Confronto (Blocco)

Il bit Z è posto ad uno se il risultato dell'istruzione CP(B) dst, src @ R è uguale alla combinazione dei bit di FLAGS codificata nel campo *cc*; poi il valore del registro specificato nel campo *cnt* è decrementato di uno e V è posto ad uno se il valore decrementato è zero. C ed S sono indefiniti.

Opzioni:

(S) Se è selezionata l'opzione S, allora dst è uguale a @ R (confronto con la *stringa*); altrimenti dst è uguale a R (confronto con la costante).

($\binom{D}{I}$) Se c'è l'opzione D, dopo il confronto il registro sorgente si *autodecrementa*; se c'è I, dopo il confronto il registro sorgente si *autoincrementa*. Il cambiamento è ± 1 per i byte, ± 2 per le parole. Se c'è l'opzione S, allora viene decrementato o incrementato anche il registro dst.

(R) Se c'è l'opzione R allora ripete l'istruzione fino a che Z è posto ad uno (verifica effettuata) oppure fino a che V è posto ad uno (*cnt* decrementato a zero); altrimenti esegue l'istruzione una volta sola.

BA + W		src @R	D/I	R	S	0
0	cnt R	dst	cc			

dst

R	@
11 + 9n	11 + 14n

Durata (cicli)

n è il numero di volte che viene eseguita l'istruzione

D/I {**1: D**, **0: I**} **(8 esadecimale)**

R: {**1: Ripete**, **0: Una volta**} **(4 esadecimale)**

S: {**1: modo dst = @**, **0: modo dst = R**} **(2 esadecimale)**

Esempio: **R0 = 20_{16}; R1 = FE_{16}; R2 = 4; R3 = $F8_{16}$; R4 = FC_{16}; R5 = 2; memoria dati: F8 = 100_{16}; FA = 5; FC = 22_{16}; FE = 20_{16}.**

CPDR R0,@R1,R2,LT ! R0 = 20_{16}; R1 = FA_{16}; R2 = 2; | 1 | 0 | **!**
Z V

CPSIR @R3,@R4,R5,EQ ! R3 = FC_{16}; R4 = 100_{16}; R5 = 0; | 0 | 1 | **!**
Z V

Il primo confronto termina perchè $20_{16} < 22_{16}$; il secondo termina quando R5 raggiunge lo zero senza che Z venga posto ad uno.

11. DAB R

C	Z	S	V	D	H
•	•	•			

(Vedasi il testo)

Correzione Decimale

Un fattore di correzione viene sommato al contenuto del registro a byte specificato supponendo che sia stata appena eseguita una ADDB o SUBB per un valore BCD privo di segno a due cifre e che sia richiesta una rappresentazione BCD a due cifre della somma o della differenza. Z ed S sono posizionati come previsto; C è posto ad uno per indicare che il risultato *decimale* della ADDB o SUBB originale era maggiore di 99 o minore di 0.

B0	dst R	0

Durata (cicli): 5

Nota: l'operazione fa uso dei bit C, D e H posti ad uno da ADDB o SUBB (o ADCB, SBCB); per cui non si possono avere operazioni che influenzino C, D o H.

Esempio: **RH0 = 27_{16}; RL0 = 69_{16}; RH1 = 14_{16}; RL1 = 38_{16}.**

```
ADDB  RL0, RL1   !RL0 = A1₁₆; RL1 = 38₁₆;
DAB   RL0        !RL0 = 7₁₆;
ADCB  RH0, RH1   !RH0 = 3C₁₆; RH1 = 14₁₆;
DAB   RH0        !RH0 = 42₁₆;
```

C	Z	S	V	D	H	
0	0	1	1	0	1	!
1	0	0	1	0	1	!
0	0	0	0	0	0	!
0	0	0	0	0	0	!

In questo caso, 2769 + 1438 = 4207, ed il valore finale zero del bit C indica che non c'è stato un superamento (cioè, il risultato reale non è 14207).

Nota speciale: Nelle CPU costruite prima del Gennaio 1980, l'istruzione DAB può lasciare S invariato.

12. $\binom{\text{DEC}}{\text{INC}}$(B) dst, #n

DEC(B), INC(B)

	•	•	•		
C	Z	S	V	D	H

(Vedasi nota)

Decremento/Incremento

Il valore destinazione è decrementato/incrementato di una quantità specificata da n ($1 \leq n \leq 16$).

modo	op	dst	#n − 1

modi dst: 0, 4, 8

dst

R	@	DA	X
4	11	13	14

Durata (cicli)

op: $\begin{cases} \textbf{2A + W: DEC(B)} \\ \textbf{28 + W: INC(B)} \end{cases}$

Note: (1) Il valore immagazzinato nell'istruzione è n −1, non n.
(2) In linguaggio di assemblaggio le istruzioni DEC(B) dst o INC(B) dst possono essere scritte senza che sia specificato il valore di n; in questo caso si assume #1.
(3) Z ed S si comportano come previsto; C non viene variato e V è posto ad uno nel caso in cui ci sia un superamento.

Esempio: **R0 = 0; R1 = 0; R2 = 100_{16}; byte della memoria dati: FE_{16} = 2; FF_{16} = 3; 100_{16} = 4.**

```
LDB RH0,@R2   !R0 = 400₁₆; R2 = 100₁₆!
DEC R2        !R2 = FF₁₆;  [0|0|0] !
                           Z S V

LDB RL0,@R2   !R0 = 403₁₆; R2 = FF₁₆!
DEC R2        !R2 = FE₁₆;  [0|0|0] !
                           Z S V

LDB RH1,@R2   !R1 = 200₁₆; R2 = FE₁₆!
LDK R3,#1     !R3 = 1!
DEC R3,#2     !R3 = −1;    [0|1|0] !
INC R3,#2     !R3 = 1;     [0|0|0] !
                           Z S V
```

13. $\binom{D}{E}$I bit mascherato

DI, EI

C	Z	S	V	D	H

Disabilitazione/Abilitazione dell'Interruzione

I bit di abilitazione dell'interruzione del FCW sono azzerati (disabilitazione)/posti ad uno (abilitazione) per ogni interruzione in cui il bit di maschera è zero nell'istruzione.

7C	0	0	E/D	MV	MNV

Durata (cicli): 7

E/D: { **1 = Abilitazione**, **0 = Disabilitazione** } **(4 esadecimale)**

MV: { **1: VI Non alterato**, **0: VI Alterato** } **(2 esadecimale)**

MNV: { **1: Non alterato**, **0: NVI Alterato** } **(1 esadecimale)**

Esempio:

```
EI VI,NVI   !VIE & NVIE posto ad 1 in FCW!
DI VI       !VIE azzerato in FCW; NVIE lasciato ad 1!
```

14. DIV(L) R, src

(Si veda testo)

Divisione

La destinazione è un numero di registri doppio della dimensione dell'argomento sorgente; una coppia di registri per DIV, quadruplo per DIVL. Il valore destinazione è diviso per il valore sorgente; il resto è immagazzinato nella metà di ordine superiore della destinazione, il quoziente nella metà di ordine inferiore.

I valori sorgente e destinazione sono numeri con segno in complemento a due. Il resto è dello stesso segno del valore destinazione.

I bit di FLAGS riportano i risultati dell'operazione. Se V = 0 allora l'operazione è stata completata normalmente; Z ed S riflettono il valore del quoziente.

Se V = 1 e Z = 0, allora il valore sorgente è zero. I valori sorgente e destinazione rimangono invariati.

Se V = 1 e Z = 1 e C = 0, allora il valore del quoziente è minore di -2^{16} oppure maggiore o uguale di 2^{16} (minore di -2^{32} o maggiore di o uguale a 2^{23} per DIVL). Il valore destinazione è indefinito.

Se V = 1, Z = 1 e C = 1 allora il valore del quoziente, q, cade nel campo $-2^{16} \leq q < -2^{15}$ o $2^{15} \leq q < 2^{16}$ ($-2^{32} \leq q < -2^{31}$ oppure $2^{31} \leq q < 2^{32}$ per DIVL). Il valore del resto è immagazzinato nella metà più significativa del valore destinazione ed il quoziente è rappresentato come un numero, in complemento a due, di 17 bit (33 bit per DIVL) costituito da S seguito dalla metà più significativa del valore destinazione.

modi src: 0, 4, 8

op: **1B: DIV**
1A: DIVL

	src R	src #	src @	src DA	src X
DIV	95	95	95	96	97
DIVL	723	723	723	724	725

Durata (cicli)

src R	src #	src @	src DA	src X
13	13	13	14	15
30	30	30	31	32

Aborto: src = 0

src R	src #	src @	src DA	src X
25	25	25	26	27
51	51	51	52	53

Aborto: src troppo piccolo

Esempio: **R0 = -1; R1 = -31; R2 = -1; R3 = -50; R4 = 2; R5 = -32768; R6 = -1.**

DIVL RQ0,RR4 !R0 = 2; R1 = 32,718; R2 = -13; R3 = -1; R4 = 2; R5 = -32768; | 0 | 0 | 1 | 0 | **!**
C Z S V

DIV RR4, R6 !R4 = ?; R5 = ?; R6 = -1; | 0 | 0 | 1 | 1 | **!**
C Z S V

Il primo caso mostra $-(15 \times 2^{33} + 50) : (5 \times 2^{15}) = -3 \times 2^{18}$ con resto -50; la seconda divisione viene abortita perchè il divisore è troppo piccolo.

Nota speciale: Nelle CPU costruite prima del Gennaio 1980, V può non essere posto correttamente ad uno, e anche la divisione per zero può causare una esecuzione impropria dell'istruzione successiva.

15. D(B)JNZ cnt R, dst

C	Z	S	V	D	H

Decremento e Salto se è Diverso da Zero

Il registro specificato nel campo cnt è decrementato di 1; se il risultato è diverso da zero, l'indirizzo *destinazione* diviene il nuovo contenuto del PC, quindi si verifica un salto indietro a dst.

F	cnt R	W	spiazzamento dst

Durata (cicli): 11

L'indirizzo destinazione viene calcolato nel seguente modo: lo spiazzamento è preso come un valore a 7 bit privo di segno (positivo), moltiplicato per 2, quindi *sottratto* dall'indirizzo della locazione che segue D(B)JNZ. (Così D(B)JNZ può essere usata per salti di indirizzi compresi nell'intervallo $ −252 a $+2).

Esempio: il codice seguente usa RH0 come contatore di loop, RH1 come accumulatore ed R1 come puntatore all'insieme di byte della memoria dati BYTAR. I quattro valori relativi a BYTAR, BYTAR + 1, BYTAR + 2 e BYTAR + 3 sono sommati in RL0 e poi il programma si ferma.

```
        LDB   RH0,#4        !Posiziona il contatore del loop!
        CLR   RL0           !Azzera l'accumulatore!
        LDA   R1,BYTAR      !Posiziona il puntatore!
LOOP:   ADDB  RL0,@R1       !Lo somma al prossimo byte!
        INC   R1            !Incrementa il puntatore!
        DBJNZ RH0,LOOP
        HALT
```

Nota Speciale: Nelle CPU costruite prima del gennaio 1980, DJNZ può non funzionare.

16. EX(B) R, src

C	Z	S	V	D	H

Scambio

I valori sorgente e destinazione vengono scambiati.

modo	2C + W	src	dst R

modi src: 0, 4, 8 (*non #*)

src			
R	@	DA	X
6	12	15	16

Durata (cicli)

Nota: I modi src sono trattati come i modi dst (nessun #), dal momento che ogni argomento è sia sorgente che destinazione.

Esempio: **R0 = 17_{16}; R15 = 100_{16}; memoria di stack: 100_{16} = 1234_{16}.**

EXB RL0,@R15 !R0 = 12_{16}; R15 = 100_{16}; 100_{16} = 1734_{16}!

17. EXTS($^{B}_{L}$) R

C	Z	S	V	D	H

Estensione del Segno

Il bit più significativo (bit di segno) della metà meno significativa del valore destinazione è copiato in ogni bit della metà più significativa. Il registro destinazione specificato ha dimensioni doppie di quelle normalmente richieste dalla parte finale del mnemonico dell'istruzione (esempio EXTSB R0, EXTS RR0, EXTSL RQ0).

B1	dst R	dimensione:

Durata (cicli): 11

dimensione:
- **0: EXTSB**
- **A: EXTS**
- **7: EXTSL**

Esempio: **R0 = −1; R1 = −50; R2 = 2.**

```
DIV   RR0,R2   !R0 = 0; R1 = -25; R2 = 2!
EXTS  RR0      !R0 = -1; R1 = -25; R2 = 2!
DIV   RR0,R2   !R0 = -1; R1 = -12; R2 = 2!
```

Come qui riportato, il risultato di una divisione è solamente la metà delle dimensioni richieste per gli argomenti dst di una istruzione di divisione; EXTS fornisce i mezzi per sostituire un argomento con un identico valore, ma costituito da un numero doppio di bit.

18. HALT

C	Z	S	V	D	H

Arresto della CPU

La CPU cessa di eseguire le istruzioni fino a quando non viene ricevuta una interruzione o un RESET (azzeramento). La CPU continua a rispondere a BUSRQ e genera una sequenza continua di cicli di rinfresco usando il campo ROW del registro di REFRESH. Siccome un ciclo di rinfresco dura tre cicli di clock la CPU può rispondere ad una interruzione ogni tre cicli di clock.

7A00

Durata (cicli): 8 + 3n

n = numero di cicli di rinfresco

19. (S)$\left(\begin{smallmatrix}\text{IN}\\ \text{OUT}\end{smallmatrix}\right)$(B) dst, src

IN(B), OUT(B), SIN(B), SOUT(B)

C	Z	S	V	D	H

Ingresso/Uscita

Viene effettuato un trasferimento di dati tra il registro specificato dalla CPU e l'indirizzo a 16-bit indicato, nello spazio di indirizzi di I/O. Il trasferimento è verso il registro della CPU per (S)IN(B) e dal registro per (S)OUT(B). S rappresenta una istruzione «speciale» di I/O; queste sono identiche a quelle normali di I/O ad eccezione dei differenti stati presenti sui piedini $ST_3 - ST_0$ (vedasi Figura 2.12); si suppone che vengano usate con il MMU.

op	I/O @R	CPU R

Durata (cicli): 10

op: $\begin{cases} \text{3C + W: IN(B); I/O @R = src; CPU R = dst} \\ \text{3E + W: OUT(B); I/O @R = dst; CPU R = src} \end{cases}$

3A + W	CPU R	op2

Durata (cicli): 12

op2: $\begin{cases} \left.\begin{matrix}\text{4: IN(B)}\\ \text{5: SIN(B)}\end{matrix}\right\} \text{CPU R = dst; modo src = DA} \\ \left.\begin{matrix}\text{6: OUT(B)}\\ \text{7: SOUT(B)}\end{matrix}\right\} \text{CPU R = src; modo dst = DA} \end{cases}$

Esempio: RL0 = 41_{16}; R1 = 1000_{16}; R2 = 1001_{16}.

```
WAIT: INB    RH0, @R1   !Preleva lo «stato» dall'indirizzo 1000₁₆ di I/O!
      TESTB  RH0        !INB non posiziona i bit di FLAGS!
      JR     PL,WAIT    !Aspetta per il bit «pronto» in bit 7!
      OUTB   @R2,RL0    !Poi invia in uscita il carattere ASCII «A» all'indi-
                        rizzo 1001₁₆ di I/O!
```

20. (S)$\binom{\text{IN}}{\text{OUT*}}$$\binom{\text{D}}{\text{I}}$(R)(B) dst@R, src@R, cnt R

	•		•		
C	Z	S	V	D	H

(Si veda il testo)

IND(B), INDR(B), INI(B), INIR(B),
SIND(B), SINDR(B), SINI(B), SINIR(B),
OUTD(B), OTDR(B), OUTI(B), OTIR(B),
SOUTD(B), SOTDR(B), SOUTI(B), SOTIR(B)

Ingresso/Uscita (Blocco)

Viene eseguito un trasferimento di dati da una locazione esterna (nello spazio indirizzo di I/O) ed una interna (nello spazio indirizzo dei dati); poi i contenuti del registro specificato nel campo *cnt* sono diminuiti di uno; V è posto ad uno se il contenuto risultante è zero. Z è indefinito.

Opzioni:

(S): Se è selezionata S, allora è una I/O speciale, altrimenti è un I/O normale (si veda la Descrizione 19).

($\binom{\text{IN*}}{\text{OUT}}$): Se è selezionata IN, la sorgente è un indirizzo esterno e la destinazione è un indizzo interno; l'opposto vale quando viene selezionata l'opzione OUT.

($\binom{\text{D}}{\text{I}}$): Se è selezionato D, dopo il trasferimento si autodecrementa il registro indirizzo interno; se è selezionata I, si autoincrementa. La variazione è ± 1 per i byte e ± 2 per le parole.

(R): Se è selezionato R, allora ripete l'istruzione fino a quando cnt = 0; altrimenti lo fa solo una volta.

3A + W		src @R	D/I	0	O/I	S
0	cnt R	dst @R	$\overline{R}$	0		

Durata (cicli): 11 + 10n
n è il numero di esecuzioni
(n = 1 se non è R)

$\overline{R}$: **0 = Ripetizione**, **1 = Una volta** **(8 esadecimale)**

D/I: **1 = Decrementa**, **0 = Incrementa** **(8 esadecimale)**

O/I: **1 = Uscita**, **0 = Ingresso** **(2 esadecimale)**

S: **1 = Speciale**, **0 = Non speciale** **(1 esadecimale)**

**Nota:* OUT è abbreviata in OT in alcuni mnemonici di assemblaggio.

Esempio: R0 = 6; R1 = 1000_{16}; R2 = FF_{16}; R3 = $100\frac{1}{16}$; byte memoria dati: FA = 21; FB = 4F; FC = 4C; FD = 4C; FE = 45; FF = 48.

```
WAIT: INB    RH4, @R1        !«Stato» dall'indirizzo 1000₁₆ di I/O!
      TESTB  RH4             !INB non posizionerà i bit di FLAGS!
      JR     PL,WAIT         !Aspetta il bit di «pronto» nel bit 7!
      OTDB   @R3,@R2,R0      !Invia in uscita un altro carattere di «HEL-
                             LO!»!
      JR     NOV,WAIT        !Se V non è uguale a uno deve arrivare qualche
                             cosa altro!
```

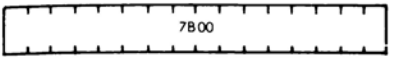
21. IRET

•	•	•	•	•	•
C	Z	S	V	D	H

(Ristabilisce le versioni salvate)

Ritorno da Interruzione

Usando il registro dello stack di sistema (RR14 o R15), viene rimossa ed eliminata una parola (la «ragione»); la parola successiva viene prelevata e posta nello FCW, infine viene prelevato un indirizzo e posto nel PC.



Z8001	Z8002
16	13

Durata (cicli)

Nota: L'indirizzo rimosso, messo nel PC, da IRET è costituito da una o due parole a seconda del tipo di segmentazione con cui funziona il processore. Bisogna fare attenzione a quando si usa lo Z8001 in modo non-segmentato, dal momento che l'indirizzo salvato nello stack dallo Z8001 quando si verifica una interruzione o una trappola è *sempre* nel formato segmentato, siccome non tiene conto del tipo di segmentazione con cui funziona il processore quando si verifica l'interruzione o la trappola.

Esempio: **RR14 = (700_{16}, FA_{16}); PC = (600_{16}, $104F_{16}$); locazione della memoria di stack nel segmento 7, inizia al valore di spiazzamento di FA_{16} : FA_{16} = $7F07_{16}$; FC_{16} = $18A0_{16}$; FE_{16} = 400_{16} ; 100_{16} = $FF4_{16}$; NSPSEG = 400_{16} ; NSPOFF = 2000_{16} .**

IRET !PC = (400_{16}, $FF4_{16}$); R15 = 2000_{16} ; FCW è posizionato alla versione non segmentata, il modo normale con

1	0	1	0	0	0
C	Z	S	V	D	H

e VI, NVI abilitati!

22. $\binom{\text{JP}}{\text{JR}}$ cc, dst

C	Z	S	V	D	H

Salto (Condizionato)

Se è vera la combinazione dei bit di FLAGS codificata nel campo *cc* allora l'*indirizzo* (NB. non il valore) della destinazione costituisce il nuovo contenuto del PC.

modo	1E	dst	cc

modi dst: 0, 4 (non 8)

JP

	@	DA	X
cc vero	10*	7	8
cc falso	7	7	8

Durata (cicli)

*** 15 nel modo segmentato**

E	cc	spiazzamento dst

JR

Durata (cicli): 6

Per JR l'indirizzo destinazione è calcolato come segue: lo spiazzamento è preso come un valore di 8 bit con segno, moltiplicato per 2, *sommato* all'indirizzo della locazione che segue JR (Quindi JR può essere usato per salti nell'intervallo **$** − 254 a **$** + 256).

Note: (1) JR non cambierà mai la parte *segmentata* dell'indirizzo.
(2) Un salto incondizionato può essere specificato in linguaggio di assemblaggio con JR dst o JP dst.

Esempio: Il seguente programma

if R0 > 5 then (azione 1) else (azione 2)

può essere implementato da

```
            CP R0,#5
            JR LE,ELSCOD
            (istruzioni per l'azione 1)
            JR EMPDSEG
ELSCOD:     (istruzioni per l'azione 2)
ENDSEG:     (continuazione del programma)
```

23. LD($^{B}_{L}$) dst, src

C	Z	S	V	D	H

(Nota: non sono alterati)

Caricamento

Il valore sorgente sostituisce il valore destinazione.

modo	op	src	dst R

modi src: 0, 4, 8

op: { 20 + W: LD(B)
14: LDL

src

	R	#	@	DA	X
LD(B)	3	7	7	9	10
LDL	5	11	11	12	13

Durata (cicli)

CARICAMENTO DI UN REGISTRO

C	dst R	#src

Durata (cicli): 5

CARICAMENTO DI UN REGISTRO
COL BYTE IMMEDIATO

modo	op	dst	src R

modi dst: 0, 4 (non 8)

op: { 2E + W: LD(B)
1D: LDL

dst

	@	DA	X
LD(B)	8	11	12
LDL	11	14	15

Durata (cicli)

CARICAMENTO DA UN REGISTRO

modo	0C + W	dst	5

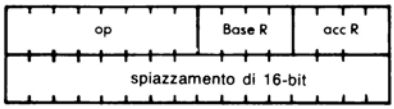
modi dst: 0, 4 (non 8)

dst

@	DA	X
11	14	15

Durata (cicli)

CARICAMENTO IN MEMORIA
DELL'IMMEDIATO

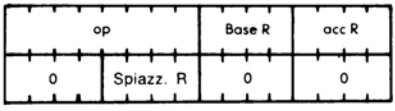


	Durata (cicli)
LD(B)	14
LDL	17

op:
- 30 + W: LD(B), 35: LDL — acc R è dst
- 32 + W: LD(B), 37: LDL — acc R è src

Base R ≠ 0; data adr = Base R + spiazzamento

CARICAMENTO CON INDIRIZZAMENTO SU BASE



	Durata (cicli)
LD(B)	14
LDL	17

op:
- 70 + W: LD(B), 75: LDL — acc R è dst
- 72 + W: LD(B), 77: LDL — acc R è src

Base R ≠ 0; data adr = Base R + spiazzamento R

CARICAMENTO INDICIZZATO SU BASE

Nota: L'assemblatore genera sempre la forma LDB R,#src.

Esempio: **R0 = 0; R1 = 100_{16}; R2 = 4; memoria dati 100_{16} = 0; 102_{16} = 22; 104_{16} = 44.**

```
LD R0, R1(#2)    !R0 = 22!
LD R0, R1(R2)    !R0 = 44!
```

24. $\binom{\text{LDA}}{\text{LDAR}}$ R, src

C	Z	S	V	D	H

Caricamento dell'Indirizzo

L'*indirizzo* (non il valore, come per la maggior parte delle istruzioni) della sorgente, sostituisce il valore destinazione. La destinazione è costituita da un registro nel modo non-segmentato e da una coppia di registri in quello segmentato.

Nel caso segmentato il bit, più significativo, ed il byte «riservato», meno significativo, della parola di segmento sono a zero per LDAR e indefiniti per LDA.

Per LDAR l'indirizzo della sorgente è valutato nel seguente modo: lo spiazzamento di 16-bit con segno viene sommato all'indirizzo della locazione successiva all'istruzione LDAR. Questo calcolo non altera la parte di segmento dell'indirizzo.

modo	36	src	dst R

modi src = solamente 4

LDA (MODI DA e X)

src

DA	X
12	13

Durata (cicli)

34	Base R	dst R
16-bit spiazzamento		

LDA (MODO DI INDIRIZZAMENTO SU BASE)

Durata (cicli): 15

74		Base R	dst R
0	Indice R	0	0

LDA (MODO INDICIZZATO SU BASE)

Durata (cicli): 15

Durata (cicli): 15

LDAR

Esempio: insieme dei caratteri del testo che si trova all'indirizzo MESG della memoria dati. La seguente codifica invierà questi caratteri in uscita usando la subroutine OUTCH.

```
        LDA     R3,MESG     !Indirizzo dell'insieme del testo in R3!
LOOP:   LDB     RL0, @R3    !Preleva il prossimo carattere del testo!
        INC     R3          !Punta al carattere seguente!
        TESTB   RL0
        JR      Z,DONE      !Carattere zero di terminazione!
        CALL    OUTCH       !Invia in uscita il carattere da RL0!
        JR      LOOP        !Ritorna indietro per eseguire il prossimo carattere!
DONE:
```

25. LDCTL(B) dst, src

C	Z	S	V	D	H

(Si veda il testo)

Caricamento dei Registri di Controllo

Il contenuto del registro di controllo specificato è trasferito allo, o caricato dall'accumulatore indicato. I bit di FLAGS vengono alterati solo se il dst è un FCW o un FLAGS (sono caricati dalla sorgente).

op	acc R	d	CTLR

Durata (cicli): 7

op:
- **8C: LDCTLB (Usato solamente con FLAGS)**
- **7D: LDCTL**

d:
- **1: acc R è src**
- **0: acc R è dst**

(esadecimale 8)

CTLR:
- **1: FLAGS**
- **2: FCW**
- **3: REFRESH**
- **4: PSAPSEG**
- **5: PSAPOFF**
- **6: NSPSEG**
- **7: NSPOFF**

Note: (1) PSAPSEG e NSPSEG possono non essére usati nello Z8002. L'assemblatore riconoscerà gli mnemonici PSAP ed NSP come PSAPOFF e NSPOFF.

(2) I bit riservati (o inutilizzati) dei registri di controllo sono inviati come zeri; essi sono ignorati quando sono inviati al registro di controllo.

Esempio: il programma deve cambiare il puntatore di indirizzo dello stato del programma per accedere alla tabella posta alla locazione NEWSTAT.

```
LDA     RR12,NEWSTAT    !Preleva l'indirizzo desiderato!
LDCTL   R11,FCW         !Salva la parola di controllo di flag!
DI      NVI,VI          !Maschera tutte le interruzioni mascherabili!
LDCTL   PSAPSEG,R12     !Posiziona il segmento!
LDCTL   PSAPOFF,R13     !Posiziona lo spiazzamento!
LDCTL   FCW,R11         !Ricostituisce FCW!
```

Questa sequenza è ancora vulnerabile se si verifica una NMI tra il momento in cui si predispone PSAPSEG e quello in cui si predispone PSAPOFF, ma protegge contro qualsiasi altra interruzione che si verifichi durante il cambiamento del puntatore di indirizzo dello stato del programma. Se si verifica una NMI nel momento critico probabilmente il programma si interromperà; si potrebbe evitare questo dando a tutte le aree di stato di programma lo stesso segmento o lo stesso spiazzamento.

26. LD($^{D}_{I}$)(R)(B) dst@R, src@R, cnt R

LDD(B), LDDR(B), LDI(B), LDIR(B)

C	Z	S	V	D	H
	•		•		

Caricamento (Blocco)

Il valore destinazione viene sostituito dal valore sorgente ed il registro specificato nel campo *cnt* viene decrementato di 1; V è posto ad 1 se il risultato del decremento ha valore 0; Z è indefinito.

Opzioni:

($^{D}_{I}$): Se c'è l'opzione D, vengono decrementati i registri src e dst; se si sceglie I, vengono autoincrementati. La variazione è ± 1 per i byte, ± 2 per le parole.

(R): Se è selezionata l'opzione R, viene ripetuta l'istruzione fino a quando cnt = 0; altrimenti l'istruzione viene eseguita una sola volta.

BA + W		src @R	D/I	1
0	cnt R	dst @R	$\overline{R}$	0

Durata (cicli): 11 + 9n.
n è il numero delle esecuzioni
(n = 1 se non è R)

D/I: { **1: Decremento** / **0: Incremento** } **(8 esadecimale)**

$\overline{R}$: { **0: Ripetizione** / **1: Una volta** } **(8 esadecimale)**

Nota: Se si sovrappongono i campi degli indirizzi sorgente e destinazione allora non funzionerà l'autoincremento e funzionerà l'autodecremento, o viceversa; la soluzione consiste nel muovere i dati *dall'*area di sovrapposizione prima di trasferirvi nuovi dati.

Esempio: **Memoria dati: $F0_{16}$ = 1000; $F2_{16}$ = 2000; $F4_{16}$ = 3000; $E0_{16}$ = 1100; $E2_{16}$ = 2200; $E4_{16}$ = 3300; R1 = $F0_{16}$; R2 = $E0_{16}$; R0 = 3.**

```
LDIR @R1,@R2,R0   !R1 = F6₁₆; R2 = E6₁₆; R0 = 0;
                  V = 1; memory: F0₁₆, E0₁₆ = 1100;
                  F2₁₆,E2₁₆ = 2200; F4₁₆,E4₁₆ = 3300!
```

27. LDK R, #n

Caricamento (di piccole) Costanti

Il valore destinazione viene sostituito dal numero n ($0 \leq n \leq 15$).

Durata (cicli): 5

Esempio:

```
LDK R0,#5   !R0 = 5!
```

Per confronto, LD R0,#5 richiede quattro byte di memoria invece dei due richiesti da LDK e richiede sette cicli invece di cinque.

28. LDM dst, src, #n

C	Z	S	V	D	H

Caricamento Multiplo

n ($1 \leq n \leq 16$) registri a parola numerati consecutivamente (mod 16). partenti da quello specificato nell'istruzione, sono caricati da o in un blocco di n parole di memoria il cui indirizzo iniziale è riportato nell'istruzione.

modo	1C	mem	op2
0	R	0	#n − 1

modi mem: 0, 4 (non #, non 8)

mem

@	DA	X
11 + 3n	14 + 3n	15 + 3n

Durata (cicli)

op2: { **9: R è src, mem è dst**
1: R è dst, mem è src.

Nota: «numerati consecutivamente (mod 16)» significa che R0 segue R15.

Esempio: **R0 = 7; R1 = 15; R2 = FA_{16}; R14 = 9; R15 = 10.**

LDM @R2,R14,#4 !Registri inalterati. Memoria dati:
FA_{16} = 9; FC_{16} = 10; FE_{16} = 7; 100_{16} = 15!

29. LDPS src

•	•	•	•	•	•
C	Z	S	V	D	H

(Caricato dal sorgente)

Caricamento dello Stato del Programma

Lo stato del programma, posto nella memoria dati partendo dall'indirizzo sorgente, viene caricato nel PC ed FCW della CPU.

Il formato dello stato del programma nella memoria dipende dallo stato di SEG in FCW *prima* che sia eseguita LDPS, trascurando lo stato di SEG caricato in FCW dall'istruzione.

Per la versione non-segmentata, il programma di stato è costituito da due parole: FCW seguito da PC. Nella versione segmentata è costituito da quattro parole: una parola zero, un FCW, poi il PC in due parole. (Vedasi Figura 2.6)

modo	39	src	0

modi src: 0, 4 (non 8, non #)

	src @	src DA	src X
non-seg.	12	16	17
seg: ind. corto		20	20
seg: ind. lungo	16	22	23

Durata (cicli)

Esempio: il programma sta funzionando in modo non-segmentato e deve passare nel modo segmentato (forse per chiamare una subroutine in un altro segmento).

```
LDAR   R0,XYZ      !Preleva il valore desiderato di PC!
PUSH   @ R15,R0    !Salvalo nello stack!
LDCTL  R0,FCW      !Preleva il valore corrente di FCW!
SET    R0,#15      !Posiziona bit 15-stato del SEG!
PUSH   @ R15,R0    !Lo mette nello stack!
LDPS   @ R15       !Carica gli stati del programma appena costruito — nel modo
                    segmentato, l'esecuzione continua in XYZ del segmento cor-
                    rente!
```

30. LDR($\frac{B}{L}$) dst, src

C	Z	S	V	D	H

Caricamento di una Costante di Programma

Viene fatto un trasferimento tra un registro specificato e una locazione dello *spazio indirizzi del programma* (non *nello spazio indirizzi dei dati*).

LDR(B)	14
LDRL	17

Durata (cicli)

op:

30 + W:	**LDR(B)**	**R è dst, la costante del programma è src**
35:	**LDRL**	
32 + W:	**LDR(B)**	**R è src, la costante del programma è dst**
37:	**LDRL**	

L'indirizzo della costante del programma è calcolata nel seguente modo: lo spiazzamento costituito da un numero di 16 bit con segno viene sommato all'indirizzo della locazione che segue l'istruzione LDR. Questa non influenzerà mai la parte di segmento dell'indirizzo.

Esempio: Si abbia un programma di diagnostica di un disco con alcuni parametri chiave assemblati in certe locazioni della memoria (piuttosto che essere sparpagliati su tutto il programma come argomenti immediati) tale da poter essere «sistemati» in un campo per adattarle alle diverse versioni del controllore o per permettere delle variabili nella configurazione di test.

```
TRACKS:   77
SECTORS:  32
    •
    •
    •
```

Una subroutine che debba far riferimento a questi parametri (che devono trovarsi nello stesso segmento) dovrebbe avere una codifica di questo tipo:

```
LDR R0, TRACKS     !R0 = 77!
LDR R1, SECTORS    !R1 = 32!
```

31. M(BIT/RES/SET)

MBIT, MRES, MSET

C	Z	S	V	D	H
	•	•			

(See text)

Test/Azzeramento/Posizionamento ad Uno nel Funzionamento Multi-Micro

MBIT controlla l'ingresso $\overline{MI}$, pone S ad uno se l'ingresso è ad uno e lascia Z inalterato. MRES e MSET azzerano e pongono ad uno l'uscita $\overline{MO}$; non alterano i bit di stato.

7B	0	op2

	Durata (cicli)
MBIT	7
MSET/MRES	5

op2:
- **A: MBIT**
- **9: MRES**
- **8: MSET**

Esempio: (vedasi la Descrizione dell'Istruzione 32: Istruzione MREQ).

32. MREQ cnt R

	•	•			
C	Z	S	V	D	H

Richiesta Multi-Micro

Gestisce il protocollo di richiesta multi-micro usando la costante del tempo di ritardo di propagazione contenuta nel registro specificato dal campo *cnt*. Pone Z ad uno se è stata tentata l'acquisizione; se l'acquisizione avviene pone S ad uno.

Il protocollo, richiede una logica esterna ed un bus per implementare una catena di priorità, funziona nel seguente modo:

(1) se $\overline{MI} = 1$ (la risorsa è già usata), pone $\overline{MO} = 0$ e termina l'istruzione con Z = 0 ed S = 0.

(2) Se $\overline{MI} = 0$ (la risorsa può essere disponibile), pone $\overline{MO} = 1$; poi decrementa cnt fino a zero (7 cicli per ogni decremento).

(3) Se $\overline{MI} = 1$ (il segnale da $\overline{MO}$ non è bloccato dalla catena di priorità), termina l'istruzione con Z = 1 e S = 1 (risorsa acquisita).

(4) Se $\overline{MI} = 0$ (il segnale era bloccato), pone $\overline{MO} = 0$ e termina l'istruzione con Z = 1 ed S = 0 (ha tentato l'acquisizione, ma è fallita).

7B	cnt R	D

Durata (cicli): 12 + 7n

n è il numero di volte che si decrementa cnt (n = 0 se non è riuscita l'acquisizione).

Esempio:

```
WAIT:   MBIT                 !Controlla MI!
        JR      MI,WAIT      !Aspetta fino a quando MI ha il valore desiderato!
        LDK     R0,#20       !Permette un ritardo di propagazione di 35 µs (a 4 MHz)!
        MREQ    R0           !Esegue il protocollo!
        JR      MI,USE       !Se lo acquisisce, lo usa!
        MRES                 !Altrimenti azzera MO!
        JR      WAIT         !Prova ancora!
USE:    (usa la risorsa)     (usa la risorsa)
        MRES                 !Azzera MO – rilascia la risorsa!
```

33. MULT(L) R, src

C	Z	S	V	D	H
•	•	•	•		

(Si veda il testo)

Moltiplicazione

La destinazione è un gruppo di registri di dimensione doppia di quella dell'argomento sorgente: una coppia di registri per MULT e quattro registri per MULTL. La metà meno significativa del valore destinazione è moltiplicata per il valore sorgente ed il prodotto sostituisce il valore destinazione; tutti i valori sono considerati come numeri segnati in complemento a due.

Z ed S vengono posizionati come previsto; C viene posto ad 1 se il prodotto è minore di -2^{15} (-2^{31} per MULTL) o maggiore o uguale a 2^{15} (2^{31} per MULTL); V viene sempre azzerato.

modo	op	src	dst R

modi src: 0, 4, 8

op: **19: MULT**
18: MULTL

src

	R	#	@	DA	X
MULT	70	70	70	71	72
MULTL	282+7n	282+7n	282+7n	283+7n	284+7n

Durata (cicli)
valore src ≠ 0

src

R	#	@	DA	X
18	18	18	19	20
30	30	30	31	32

Durata (cicli)
valore src = 0

n è il numero di bit diversi da zero nell'*ampiezza*, della metà meno significativa, del valore destinazione.

Esempio: **R0 = 99; R1 = 99; R2 = −2; R3 = 0; R4 = 1; R5 = 16.**

MULTL RQ0,RR4 !R0 = −1; R1 = −3; R2 = −32; R3 = 0;
R4 = 1; R5 = 16; | 1 | 0 | 1 | 0 | **!**
C Z S V

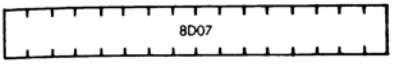
In questo esempio, $-2^{17} \times (2^{16} + 2^4) = -(2^{33} + 2^{21})$; notate che sono irrilevanti i valori originali di R0 e R1. Il tempo richiesto per l'esecuzione di questa istruzione è 282 + 7 = 289 cicli, siccome 2^{17} non ha bit zero.

Nota Speciale: Nelle CPU costruite prima di Marzo 1980 l'istruzione MULTL può non funzionare quando è abilitato il funzionamento del registro REFRESH.

34. NOP

Nessuna Operazione

La CPU non fa niente per 7 cicli.



Durata (cicli): 7

Nota: NOP ha due usi principali: fornire la struttura portante per i loop di attesa e riempire il posto lasciato dalle istruzioni eliminate.

Esempio:

```
LP:  NOP              !7  cicli!
     NOP              !7  cicli!
     NOP              !7  cicli!
     NOP              !7  cicli!
     DJNZ  R0,LP      !11 cicli!
```

Questo loop verrà eseguito in 39n cicli, dove n è il valore iniziale di R0, considerato come un numero positivo *privo di segno* compreso tra 1 e 2^{16} (esempio R0 = 0 significa che il loop sarà eseguito 2^{16} volte). Usando un clock di 4 MHz questo permette un'intervallo compreso tra 9,75 μs e 639 ms; un numero maggiore di NOP potrebbe dare un ritardo massimo maggiore, ma con una regolarizzazione piuttosto grossolana.

35. POP(L) dst, @R

C	Z	S	V	D	H

Prelievo dallo Stack

Il valore della locazione indirizzata dal registro sorgente (puntatore dello stack) sostit. ..e il valore destinazione; poi il registro sorgente viene incrementato di una quantità pari al numero di byte trasferiti (2 per POP, 4 per POPL). (Vedasi Figura 2.2).

modo	op	src @R	dst

modi dst: 0, 4, 8

	dst			
	R	@	DA	X
POP	8	12	16	16
POPL	12	19	23	23

Durata (cicli)

op: { **17: POP**
15: POPL

Note: (1) Per POPL il puntatore di stack usato nel campo src non può essere usato anche nel campo dst. Esempio POPL RR4, @ R4 è illegale.

(2) L'autoincremento del registro sorgente viene fatto per *ultimo*, dopo tutti i calcoli di indirizzo e trasferimento dati.

Esempio: **R6 = FA_{16}; Memoria dati FA_{16} = 1111; FC_{16} = 2222; FE_{16} = 3333.**

```
LD    R0,2(R6)      !R0 = 2222; R6 = FA16; memoria invariata!
POP   2(R6),@R6     !R0 = 2222; R6 = FC16; FA16 = 1111; FC16 = 1111;
                     FE16 = 3333!
```

36. PUSH(L) @R, src

C	Z	S	V	D	H

Deposito nello Stack

Il registro destinazione (puntatore di stack) è decrementato di una quantità pari al numero di byte da trasferire (2 per PUSH, 4 per PUSHL); poi il valore sorgente sostituisce il valore della locazione di memoria indirizzata dal nuovo valore del registro destinazione. (Vedasi Figura 2.2).

modo	op	dst @R	src

modi src: 0, 4, 8
(non #)

op: { **13: PUSH**
11: PUSHL

	R	@	DA	X
PUSH	9	13	14	14
PUSHL	12	20	21	21

Durata (cicli)

PUSH(L)—src non #

0D	dst @R	9

Durata (cicli): 12

PUSH dell'immediato

Note: (1) Per PUSHL il registro puntatore di stack usato nel campo dst non può essere usato anche nel campo src; esempio PUSHL @RR6, @RR6 è illegale.

(2) L'autodecremento del registro destinazione viene effettuato dopo tutti i calcoli di indirizzo, ma prima del trasferimento dati.

Esempio: R6 = FC_{16}; R0 = 2222; Memoria dati FA_{16} = 0000; FC_{16} = 1111; FE_{16} = 3333.

```
PUSH @R6,@R6   !R6 = FA16; memoria FA16 = 1111; FC16 = 1111;
               FE16 = 3333!
LD   2(R6),R0  !R6 = FA16; R0 = 2222; memoria FA16 = 1111;
               FC16 = 2222; FE16 = 3333!
```

37. RET cc

C	Z	S	V	D	H

Ritorno da Subroutine

Se è vera la combinazione dei bit di FLAGS codificata nel campo cc, allora il PC (2 parole se siamo nel modo segmentato, 1 se siamo in quello non-segmentato) viene prelevato dallo stack controllato dal registro di stack implicito (RR14 o R15).

9E	0	cc

	non-seg.	seg.
cc Falso	7	7
cc Vero	10	13

Note: (1) L'uso di questa istruzione prevede una precedente istruzione, CALL, CALR o equivalente che metta, nel formato opportuno, un indirizzo nello stack.

(2) RET può essere scritto senza il campo cc, in questo caso il ritorno viene eseguito senza verificare alcuna condizione.

Esempio: I numeri nel campo commenti indicano la sequenza delle istruzioni eseguite.

```
        CALL XYZ    !1!
        HALT        !4!
XYZ:    NOP         !2!
        RET         !3!
```

38. R($^{R}_{L}$)(C)(B) R, #posizioni

RL(B), RLC(B), RR(B), RRC(B)

•	•	•	•		
C	Z	S	V	D	H

(Si veda sotto)

Rotazione a Destra/Sinistra

Il registro destinazione è considerato come un insieme circolare di registri di un bit in cui il bit più significativo (7 o 15) è spostato di una posizione alla destra del bit zero. Se C è incluso viene trattato come un ulteriore registro ad un bit posto tra il bit zero e quello più significativo. Il circolo viene ruotato verso sinistra o destra di un numero di posti pari al valore specificato (1 o 2).

Se C non è incluso viene posizionato all'ultimo valore utilizzato per passare dai bit più significativi a quelli meno significativi.

B2 + W	dst R	C	R/L	p	0

1	2
Posizioni	
6	7
Durata (cicli)	

C: { **1 = Comprende C**, **0 = Non comprende C** } **(8 esadecimale)**

R/L: { **1 = Destra**, **0 = Sinistra** } **(4 esadecimale)**

p: { **1 = 2 Posizioni**, **0 = 1 Posizioni** } **(2 esadecimale)**

Note: (1) Viene modificato come detto sopra; Z ed S sono posizionati in funzione del risultato; V è posto ad uno se cambia il segno del valore destinazione.

(2) Se nella designazione del linguaggio di assemblaggio viene ommesso il numero di posizioni di cui si deve eseguire la rotazione, si assume il valore 1.

Esempio: **R0 = $AF05_{16}$.**

RLB	**RH0,#2**	**!R0 = $BE05_{16}$;**	0	0	1	0	!
RR	**R0**	**!R0 = $DF02_{16}$;**	1	0	1	0	!
RLCB	**RL0,#2**	**!R0 = $DF0A_{16}$;**	0	0	0	0	!
RRCB	**RH0**	**!R0 = $6F0A_{16}$;**	1	0	0	1	!
RLC	**R0,#2**	**!R0 = $BC2A_{16}$;**	1	0	1	1	!
			C	Z	S	V	

39. R($\frac{L}{R}$)DB dst R, src R

RLDB, RRDB

C	Z	S	V	D	H
	•	•			

Rotazione di Digit tra Byte

I due digit del registro sorgente, ed il digit meno significativo del registro di destinazione, vengono considerati come una struttura circolare di digit e ruotati di una posizione verso sinistra o destra.

Durata (cicli): 9

op: { **BE: RLDB**
BC: RRDB

Nota: I registri a byte sorgente e destinazione non devono essere gli stessi.

Esempio: **R0 = ABCD$_{16}$; R1 = EF12$_{16}$.**

RLDB RL1,RH0	**!R0 = B2CD; R1 = EF1A;**	0	0	!
RLDB RH0,RL0	**!R0 = BCD2; R1 = EF1A;**	0	1	!
RLDB RL0,RH1	**!R0 = BCDE; R1 = F21A;**	0	1	!
RLDB RH1,RL1	**!R0 = BCDE; R1 = F1A2;**	0	1	!
RRDB RL1,RL0	**!R0 = BC2D; R1 = F1AE;**	0	1	!
RRDB RH0,RH1	**!R0 = B12D; R1 = CFAE;**	0	1	!
		Z	S	

40. SC # indice

•	•	•	•	•	•
C	Z	S	V	D	H

(Posizionato da PSA)

Chiamata Sistema

Quando si verifica la trappola di chiamata sistema, l'istruzione diventa la *ragione*. Il campo indice, nel byte meno significativo della ragione, consente una selezione delle routine software analoga all'interruzione vettoriale.

7F	# indice

non-seg.	seg.
33	39

Durata (cicli)

Esempio: **FCW = 8000_{16}; PC = (300_{16},200_{16}); RR14 = (500_{16},100_{16}); Puntatore dello stack di sistema = (600_{16},$B0_{16}$).**

SC #7 **!FCW,PC Caricato da PSA (Vedasi Figura 2.9)**
NSPSEG = 500_{16}; NSPOFF = 100_{16}; RR14 = (600_{16},BA_{16})*;
Memoria dello stack nel segmento 6 = $7F07_{16}$; BC_{16} = 300_{16};
BE_{16} = 202_{16}!

***Si suppone che FCW riceva da PSA il modo sistema = 1**

41. $S\left(\begin{smallmatrix}D\\L\\R\end{smallmatrix}\right)\left(\begin{smallmatrix}A\\L\end{smallmatrix}\right)\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right)$ R, src

C	Z	S	V	D	H
•	•	•	•		

$SDA\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right)$, $SDL\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right)$, $SLA\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right)$, $SLL\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right)$, $SRA\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right)$, $SRL\left(\begin{smallmatrix}B\\L\end{smallmatrix}\right)$

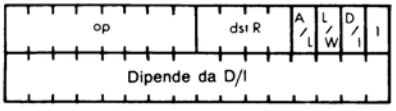
Spostamento

I contenuti del registro destinazione sono spostati secondo la direzione ed il numero di posizione specificate nel valore sorgente.

Per uno spostamento a sinistra, le posizioni libere, meno significative vengono riempite con zeri, mentre l'ultimo bit spostato dall'estremo più significativo viene usato per posizionare C. Per lo spostamento a destra, C viene posizionato dall'ultimo bit spostato dall'estremità meno significativa. I bit più significativi liberi sono riempiti con zeri per uno spostamento logico o con il bit del segno originale per uno spostamento aritmetico.

Un valore sorgente negativo indica uno spostamento a destra, un valore positivo indica uno spostamento a sinistra; l'ampiezza è il numero di bit da spostare. Se l'ampiezza supera il numero di bit contenuti nel registro destinazione, l'operazione è indefinita; se è zero non si ha alcun spostamento.

Z ed S sono posizionati in funzione del valore di destinazione finale (anche per un valore sorgente zero); C viene posizionato come detto sopra; V è indefinito per gli spostamenti logici, azzerato per gli spostamenti aritmetici a destra ed uguale a uno per gli spostamenti aritmetici a sinistra se i valori iniziali e finali di destinazione hanno segno diverso.



Dinamico	15 + 3n
Immediato	13 + 3n

n = numero di posizioni spostate o 1 se non c'è spostamento

Durata (cicli)

op: { **B2:** (L/W = 0): **byte**; **B3:** { L/W = 1: **Parola lunga**; L/W = 0: **Parola** }

(hex 4)

A/L: { **1: Aritmetico**; **0: Logico** }

(hex 8)

D/I: { **1: Dinamico; formato dalla 2ª parola**; **0: Immediato; la 2ª parola contiene il valore sorgente** }

(2 esadecimale)

0	src R	0	0

Nota: Il valore sorgente di SR $\binom{A}{L}\binom{B}{L}$ viene specificato nel linguaggio di assemblaggio come positivo, ma nell'istruzione viene assemblato un valore negativo; la coerenza con l'uso delle altre istruzioni avrebbe richiesto:

SA R0,#−5 invece di SRA R0,#5
SA R0,#5 invece di SLA R0,#5
SA R0,R1 invece di SDA R0,R1

Tale utilizzo sarebbe stato troppo sconveniente e soggetto ad errore.

42. TCC(B) cc, R

C	Z	S	V	D	H

Test del Codice Condizione

Se è vera la combinazione dei bit di FLAGS codificata nel campo *cc*, il bit meno significativo del valore destinazione è posto ad 1, altrimenti il bit meno significativo *non* viene *cambiato* (NB.: *non* posto a zero).

AE + W	dst R	cc

Durata (cicli): 5

Esempio:

1	0	0	1
C	Z	S	V

; **R0 = 0.**

```
TCC LE,R0     !R0 = 0001₁₆;  LE è vera!
TCC GT,R0     !R0 = 0001₁₆;  GT è falsa — ma non azzera R0!
CLR R0        !R0 = 0!
TCC GT,R0     !R0 = 0!
```

43. TEST($^{B}_{L}$) dst

	•	•	•		
C	Z	S	P	D	H

(Si veda la nota)

Test

Pone ad 1 Z se il valore destinazione è 0, S se il bit più significativo è 1,P se TESTB ed il valore destinazione hanno parità pari.

modo	op	dst	op2

modi dst: 0, 4, 8

	R	@	DA	X
TEST(B)	7	8	11	12
TESTL	13	13	16	17

Durata (cicli)

op: { **0C + W: TEST(B); op2 = 4**
1C: TESTL; op2 = 8

Nota: P è lasciato inalterato da TEST, indefinito per TESTL

Esempio: R0 = FF07.

TEST R0 ! 0 1 !
Z S

TESTB RH0 ! 0 1 1 !
TESTB RL0 ! 0 0 0 !
Z S P

44. TR(T)$\binom{D}{I}$(R)B string @R, table @R, cnt R

TRDB, TRDRB, TRIB, TRIRB,
TRTDB, TRTDRB, TRTIB, TRTIRB

	•		•		
C	Z	S	V	D	H

Conversione (Blocco)

Il byte indirizzato dal registro *string* è usato come indice alla tabella di byte il cui indirizzo base si trova nel registro *table*. Il byte così indirizzato è chiamato *conversione* del byte originale; è memorizzato in una delle due posizioni funzioni di (T).

Poi i contenuti del registro *cnt* sono decrementati di uno e V è posto ad uno se il contenuto risultante è zero.

Opzioni:

(T): Se si ha l'opzione T, allora la conversione è memorizzata nel registro a byte RH1 e Z è posto ad uno se il valore convertito è zero; altrimenti la conversione sostituisce il byte originale indirizzato dal registro *string; RH1 e Z sono indefiniti.*

($\binom{D}{I}$): Se si ha l'opzione D, autodecrementa il registro string; se si ha I lo autoincrementa; il cambiamento è di ± 1.

(R): Se si ha l'opzione R l'istruzione viene ripetuta fino a quando *cnt* è decrementato a zero o (solamente per T) RH1 ottiene un valore di conversione *diverso da zero.*

B8			string @R	D/I	R	T	0
0		cnt R	table @R	0			

Durata (cicli): 11 + 14n
n è il numero di esecuzioni; n = 1 se non è R.

D/I: $\begin{cases} \mathbf{1} = \textbf{Decremento} \\ \mathbf{0} = \textbf{Incremento} \end{cases}$ **(8 esadecimale)**

R: $\begin{cases} \mathbf{1} = \textbf{Ripete} \\ \mathbf{0} = \textbf{Una volta} \end{cases}$ **(4 esadecimale)**

: $\begin{cases} \mathbf{1}: \quad \textbf{dst} = \textbf{RH1; } \textit{Test} \textbf{ per} = \mathbf{0} \\ \mathbf{0}: \quad \textbf{dst} = \textbf{@string(Non fa il test)} \end{cases}$ **(2 esadecimale)**

Nota: Viene trascurata la designazione, nel linguaggio di assemblaggio, @ table. L'argomento prelevato dalla memoria usa table in una variante dell'indirizzamento indicizzato tramite base: table (@ string). La memorizzazione dell'argomento (nel caso non T) è realmente un indirizzo indicizzato tramite base; table (string).

Esempio: Supponete sia data una tavola nella memoria dati costituita da 128 punti di ingresso di un byte: per ciascun i, $0 \leq i \leq 127$, l'entrata i-esima è il codice EBCDIC del carattere il cui codice ASCII è i (EBCDIC è un altro codice carattere). L'indirizzo della tabella è ASCEBC. Supponete di avere una stringa di 80 caratteri ASCII che inizia nella memoria dati a CHARSTR.

```
LD      R3,#80          !Lunghezza della stringa!
LDA     R2,ASCEBC       !Indirizzo della tabella!
LDA     R1,CHARSTR      !Indirizzo della stringa!
TRIRB   @R1,@R2,R3      !La stringa è convertita in EBCDIC!
```

45. TSET(B) dst

		•			
C	Z	S	V	D	H

Test e Posizionamento ad Uno

Viene controllato il segno del valore destinazione e S posizionato coerentemente; poi il valore destinazione viene sostituito con −1.

modo	OC + W	dst	6

modi dst: 0, 4, 8

R	@	DA	X
7	11	14	15

Nota: Questa istruzione è fornita per implementare dei *semafori:* indicatori di controllo per le risorse condivise. La convenzione stabilita è

positivo = libero

negativo = utilizzato

$\overline{\text{BUSRQ}}$ non viene trattato durante l'esecuzione di questa istruzione, per cui *se c'è solamente una CPU nel* sistema, non ci possono essere altri accessi al semaforo tra il controllo e il posizionamento.

Esempio:

```
WAIT: TSET  SEMA      !Aspetta che SEMA diventi positivo!
      JR    MI, WAIT
      (usa la risorsa)
      CLR   SEMA      !Rilascia la risorsa!
```

La risorsa condivisa in questo esempio potrebbe essere un insieme di indicatori di programma, di dati o potrebbe essere l'accesso ai registri di controllo della CPU.

Capitolo V

Modi di Indirizzamento dello Z8000

Nel Capitolo IV abbiamo descritto tutte le istruzioni dello Z8000 ed abbiamo citato i modi di indirizzamento che esse usano per indicare le locazioni dei loro argomenti. In questo capitolo presenteremo questi modi di indirizzamento.

Le istruzioni prelevano i loro argomenti da tre posti diversi: dall'istruzione stessa, da un registro o da una locazione di memoria. Se una istruzione preleva il suo argomento da una locazione di memoria, l'indirizzo di tale locazione può essere trovato in uno dei seguenti tre modi: nell'istruzione stessa, in un registro o per mezzo di un calcolo. Il calcolo usato per determinare gli indirizzi è sempre costituito da una somma di un indirizzo con uno spiazzamento. Esistono tre varianti: sia l'indirizzo che lo spiazzamento possono essere nei registri, l'indirizzo può trovarsi in un registro e lo spiazzamento in una istruzione, lo spiazzamento può essere in un registro e l'indirizzo in una istruzione. La Figura 5.1 illustra queste tre possibilità e i simboli associati ai corrispondenti modi di indirizzamento.

La Figura 5.1 non fa distinzione tra argomenti sorgente e destinazione; ad eccezione di quando l'argomento è compreso nell'istruzione; ossia solo quando è un argomento sorgente. Tutti gli altri modi rappresentanti possono essere utilizzati sia per argomenti sorgente che per quelli destinazione.

Non è difficile capire perchè esistono i modi di indirizzamento immediato (#), mediante registro (R), con indirizzo diretto (DA) e indiretto su registro (@); ma, perchè ci sono i modi di indirizzamento B, X e BX? Nei primi calcolatori c'era solo il modo X. Per esempio, sull'IBM 7094 uno scriveva l'equivalente di

```
LD R0,TABLE(R1)
```

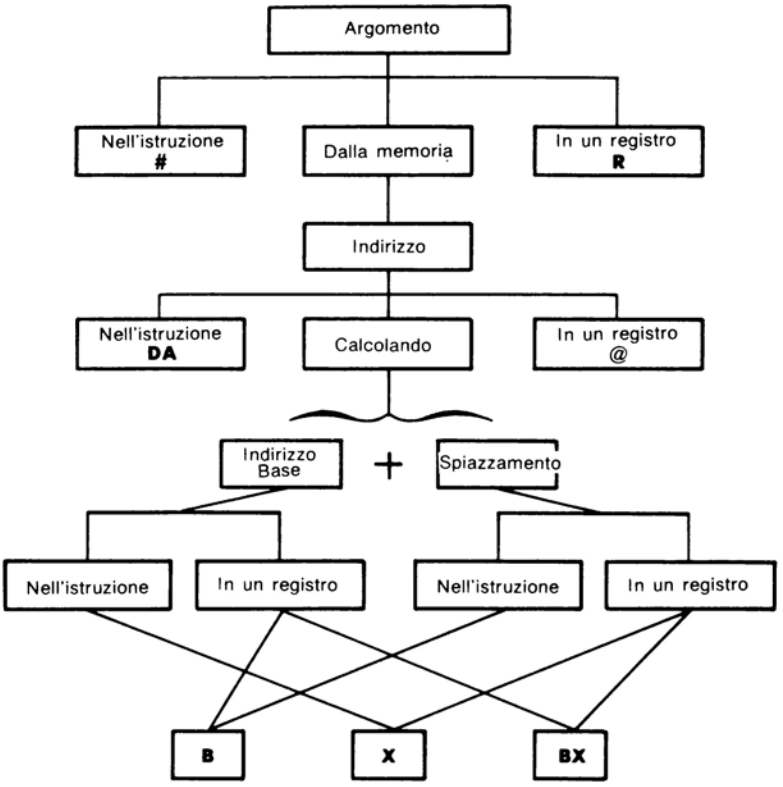


Figura 5.1 — Modi di Indirizzamento dello Z8000

e in quel caso, R1 non era un registro non specializzato, ma era un registro indice. Esso aveva 15 bit mentre l'accumulatore ne aveva 36 e i campi di indirizzo delle istruzioni (non c'erano registri indirizzo) avevano 18 bit. L'indirizzo TABLE era contenuto nell'istruzione.

Il nuovo uso dell'indirizzamento indicizzato si sviluppò con l'arrivo dei successivi calcolatori dotati di molti registri non specializzati. In questo caso, gli elementi dello stack possono essere indirizzati come 0(R), 2(R), 4(R), ecc. se un registro (R) non specializzato può essere considerato come registro indice *oppure* come puntatore di stack. (Si vedano gli esempi per le Descrizioni Istruzioni 35 · POP e 36 · PUSH). Questo si verifica usando il modo indicizzato all'inverso, infatti 0, 2, 4, ecc., dovrebbero essere indirizzi ed R dovrebbe contenere lo spiazzamento, invece in realtà R contiene l'indirizzo e 0, 2, 4, ecc., sono gli spiazzamenti. Questo non crea differenze se gli indirizzi e gli spiazzamenti sono indistinguibili gli uni dagli altri, ma questo non è vero su uno Z8000 *segmentato*. Questo è il mo-

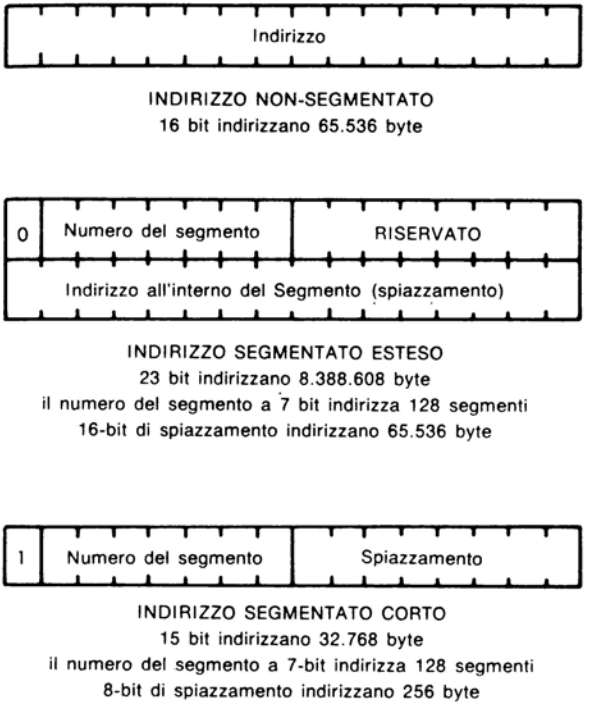


Figura 5.2 — Formati degli Indirizzi Usati nelle Istruzioni

tivo per cui il modo di indirizzamento *su base* (B); rende ancora possibile l'espressione 2(R), sebbene B sia scritto come R(#2). Quindi si mantiene l'uso originale: l'indirizzo base fuori dalle parentesi e lo spiazzamento dentro.

Il prossimo punto riguarda l'indirizzamento in modo indicizzato su base (BX): qualche volta nè l'indirizzo base nè lo spiazzamento è conosciuto fino al momento dell'esecuzione dell'istruzione. (esempio, vedasi Figura 3.9).

Nel Capitolo II abbiamo trattato il formato del registro degli indirizzi segmentati (vedasi Figura 2.3). Come mostrato in Figura 5.1 esistono due diversi modi (DA e X) per rappresentare un indirizzo nell'istruzione e ciascuno ha tre formati: uno per gli indirizzi non-segmentati e due per gli indirizzi segmentati. Questi formati sono riportati in Figura 5.2.

Questo significa che ogniqualvolta che il campo di modo, che controlla sia src che dst, contiene il valore 4 (vedasi Figura 4.16), nella memoria programma, all'istruzione base, deve seguire un indirizzo espresso in uno dei tre formati di Figura 5.2. La Figura 5.3 riporta alcuni esempi a riguar

CPL RR2,X(R6) (Non-Segmentato)

CPL RR2,X(R6) (Segmentato, Indirizzo esteso)

CPL RR2,X(R6) (Segmentato, Indirizzo corto)

CPL RR2,X (Non-Segmentato)

CPL R,src
(Istruzione Base usata sopra)

Figura 5.3 — Indirizzi nelle Istruzioni

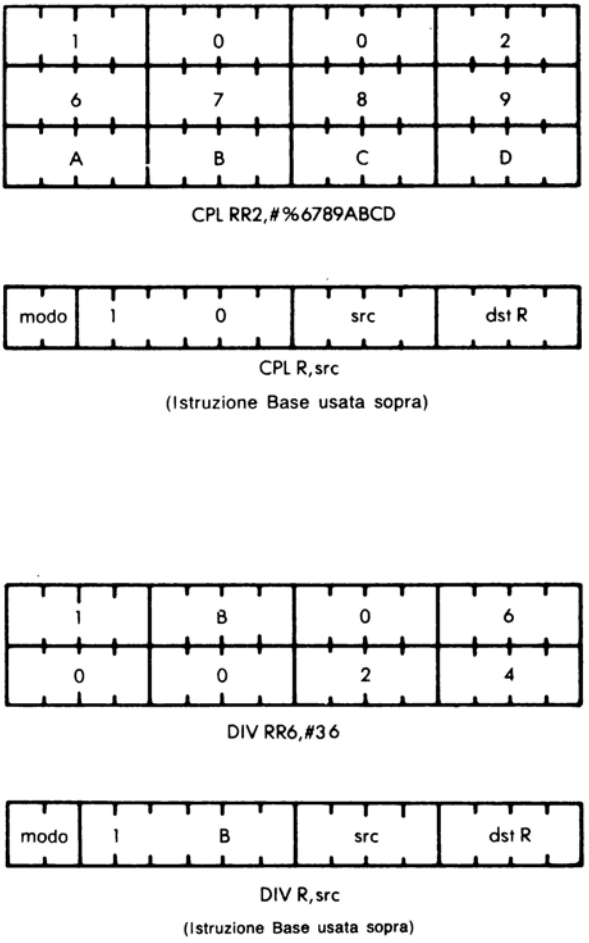
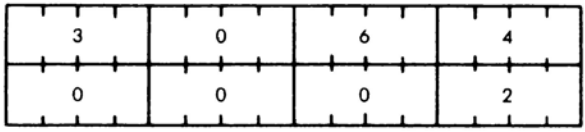


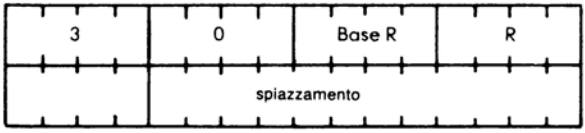
Figura 5.4 — Esempi di Argomenti Immediati

do. Si noti come il formato dell'istruzione, costruita al momento dell'assemblaggio, debba corrispondere esattamente al modo segmentato fissato al momento dell'esecuzione. Questo non si verifica (eccetto per i registri di numero pari) con i *formati* dell'istruzione relativi agli altri cinque modi di indirizzamento.

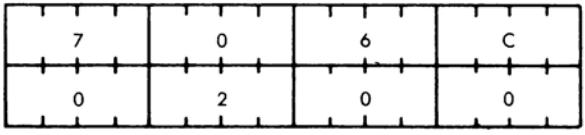
Degli altri tre formati «ordinari» (R, @, #) solamente # richiede della memoria addizionale oltre all'istruzione di base; infatti ogniqualvolta che appare la combinazione di un modo zero con R0 nel campo controllato src (vedasi Figura 4.16), il valore sorgente viene appeso all'istruzione base. Alcuni esempi di questo caso sono riportati nella Figura 5.4.



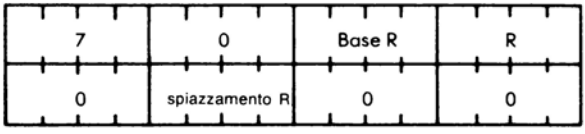
LDB RH4,R6(#2) (Non-Segmentato)
LDB RH4,RR6(#2)(Segmentato)



LDB R, Base R(# spiazzamento)
(Formato dell'istruzione usata sopra)



LDB RL4,R6(R2) (Non-Segmentato)
LDB RL4,RR6(R2) (Segmentato)



LDB R, Base R (Spiazzamento R)
(Formato dell'istruzione usata sopra)

Figura 5.5 — Esempi di Modo di Indirizzamento su Base e Indicizzato su Base

Gli altri due modi di indirizzamento, B e BX, sono disponibili solamente con le istruzioni LD e LDA. Nelle corrispondenti descrizioni delle istruzioni sono illustrati i formati completi, compreso lo spiazzamento per l'istruzione B.

La Figura 5.5 riporta alcuni esempi. Si noti che, se la CPU sta funzionando in modo segmentato o non-segmentato la stessa istruzione ha due significati diversi.

ESADEC.	SIGNIFICATI	ESADEC.	SIGNIFICATI
0	**R0,RH0,RR0,RQ0**	8	**R8,RL0,RR8,RQ8**
1	**R1,RH1**	9	**R9,RL1**
2	**R2,RH2,RR2**	A	**R10,RL2,RR10**
3	**R3,RH3**	B	**R11,RL3**
4	**R4,RH4,RR4,RQ4**	C	**R12,RL4,RR12,RQ12**
5	**R5,RH5**	D	**R13,RL5**
6	**R6,RH6,RR6**	E	**R14,RL6,RR14**
7	**R7,RH7**	F	**R15,RL7**

Figura 5.6 — Interpretazione dei Campi di Registro delle Istruzioni

Esiste un altro punto importante: ogniqualvolta, il registro di stack (R15 o RR14) viene usato come un registro indirizzo, lo stato di uscita dalla CPU assume il valore di «riferimento allo stack», esempio nei modi @,e B quando è nel modo *base* o BX. Quando il registro di stack è usato come un indice (esempio, 2(SR)), le linee di stato indicano un riferimento ad un dato in memoria.

Si noti anche la codifica esadecimale dei registri a byte: RH0 ad RH7 sono numerati da 0 a 7; RL0 e RL7 sono numerati da 8 a F. La Figura 5.6 mostra tutte le codifiche dei registri usate nelle istruzioni. Ad eccezione di RL0, ..., RL7 ciascun indirizzo è codificato con il corrispettivo numero assegnato nel linguaggio assembler. Perciò, Rn, RRn, RQn e RHn appaiono tutti come n nel campo registro; RLn appare come 8 + n.

Esiste un ulteriore modo di indirizzamento, studiato per accedere allo spazio di indirizzo programma: *l'indirizzamento relativo.* L'indirizzamento di modo relativo ricava un indirizzo sommando o sottraendo dal PC uno spiazzamento contenuto nell'istruzione. Esistono molte varianti di questo modo tutte riportate nelle descrizioni delle varie istruzioni (CALR, JR, LDAR, LDR). L'indicazione dell'uso di questo modo viene data nello mnemonico dell'assembler. Occorre ricordare che il valore del PC usato nell'indirizzamento relativo è quello della locazione che segue l'istruzione. Le varie descrizioni di istruzioni fanno corrispondere questo «$,» con il contatore di locazione. Per cui, nelle descrizioni delle istruzioni la «finestra» relativa all'indirizzamento relativo è espressa in termini di indirizzo dell'istruzione. Ma se voi state assemblando normalmente una di queste istruzioni, dovete ricordarvi di calcolare lo spiazzamento partendo dall'indirizzo dell'istruzione successiva.

Questo conclude la nostra descrizione dei modi di indirizzamento e delle istruzioni dello Z8000. La parte rimanente del libro spiega come usarli.

Capitolo VI

Tecniche per la gestione dell'Ingresso/Uscita (Input/Output)

La CPU Z8000 è un dispositivo a 40 o 48 piedini; tutte le sue comunicazioni con i dispositivi esterni devono avvenire attraverso questi piedini. Tutti gli I/O dipendono dagli zeri ed uni presenti sui piedini del bus indirizzi/dati ($AD_{15} - AD_0$), sui tre piedini del bus di controllo ($\overline{AS}$, $\overline{MREQ}$, $\overline{DS}$), sui sette piedini di stato ($R/\overline{W}$, $B/\overline{W}$, $N/\overline{S}$, $ST_3 - ST_0$) e sui tre piedini delle interruzioni ($\overline{NMI}$, $\overline{NVI}$, $\overline{VI}$).

Naturalmente, tutte le operazioni interne alla CPU dipendono dagli zeri e uni che si trovano nei vari registri e nei vari bus, ma l'evoluzione delle CPU negli anni ci ha portati sempre più lontani da questi zeri ed uni grazie alla possibilità di operare a livelli di astrazione sempre più elevati. Allo stesso modo, ogni nuova generazione di microprocessori è in grado di usare I/O più sofisticati dei semplici ritardi e impulsi generati sulle singole linee. Tuttavia, questi nuovi sviluppi sono stati forniti principalmente sotto forma di dispositivi *opzionali* di supporto *esterno*. Questo significa che si possono omettere quando si considerano principalmente i costi e il conteggio dei componenti. E anche quando sono inclusi, il fatto che siano componenti esterni significa che il programmatore è maggiormente coinvolto nei dettagli di I/O rispetto a quando questi non sono presenti.

Per queste ragioni, in questo capitolo discutiamo alcune delle tecniche tradizionali di I/O come i ritardi, gli impulsi, il trattamento delle stringhe seriali di bit, il controllo delle temporizzazioni e similari. Nel prossimo capitolo discuteremo i componenti speciali che rendono obsolete queste tecniche.

Per mezzo di un *bus* vengono rese disponibili ai dispositivi di I/O i sedici piedini indirizzi/dati delle CPU Z8000. Analogamente al significato della parola latina *omnibus* da cui deriva, il bus può essere usato per ogni

dispositivo. Come per una strada di transito, esso collega tutti i dispositivi del sistema ed i segnali possono essere inviati ad una qualsiasi connessione. Il bus I/O comprende anche le tre linee del bus controllo e le sette linee del bus di stato menzionate prima. Queste linee del bus di controllo e di stato non sono direttamente sotto il controllo del programmatore; la CPU Z8000 invia i segnali su di loro come prestabilito.

Il programmatore controlla le linee indirizzi/dati mediante le istruzioni di I/O (vedasi Descrizioni 19 e 20). Per esempio, il comando

```
INB RL0, %1E
```

fa in modo che la CPU Z8000 esplichi il protocollo di bus necessario per ottenere un byte di dato dal dispositivo di I/O che risponde all'indirizzo $1E_{16}$. In questo luogo non ci riguarda la sequenza reale dei segnali su $AD_{15} - AD_0$, $\overline{MREQ}$, $\overline{AS}$, $\overline{DS}$, $B/\overline{W}$, $R/\overline{W}$, ed $ST_3 - ST_0$. Tutto quello che ci interessa conoscere è che il dispositivo, a cui noi siamo interessati, ci invierà un byte di dato se lo indirizziamo come $1E_{16}$.

Ci sono alcuni punti che vanno notati per quanto riguarda gli indirizzi di I/O tipo 1E sopra riportato: essi costituiscono uno spazio indirizzi completamente separato dalla memoria indirizzi ($\overline{MREQ}$ viene usato per distinguere una richiesta di memoria da una richiesta di I/O); e possono avere valori compresi tra 0 e FFFF (64K), ma non ne hanno bisogno. Per esempio, il dispositivo che risponde a 1E può anche rispondere a 5E, DE, FF1E, ecc., se si stanno considerando solamente i 6 bit meno significativi di $AD_{15} - AD_0$; esempio, ad $AD_5 - AD_0$.

Prendiamo l'esempio più semplice possibile di I/O: un interruttore. Assumiamo che quando il bit zero dell'indirizzo FURN di I/O è posto ad uno, questo determini la chiusura di un interruttore, mentre uno zero lo apra. Per renderlo interessante, immaginate che l'interruttore controlli il vostro forno. Perciò,

```
LDB RH0,#1
OUTB FURN,RH0
```

accenderà il vostro forno;

```
CLRB RH0
OUTB FURN,RH0
```

lo spegnerà. Il segnale generato da ciascuno di questi gruppi di istruzioni è chiamato *livello*. Il segnale generato da uno dei due gruppi seguito dal-

l'altro è chiamato *impulso*. Il tempo trascorso tra i due è la durata dell'impulso. Per esempio, potrete usare la seguente sequenza:

```
LDB RH0,#1
OUTB FURN,RH0
(attesa di 20 minuti)
CLRB RH0
OUTB FURN,RH0
```

per inviare un impulso della durata di 20 minuti al vostro forno. L'«attesa di 20 minuti» può essere realizzata nel seguente modo:

```
        LDL RR0,# -240000000
X:      ADDL RR0,#1
        JR NZ,X
```

ESERCIZIO 1: (a) Verificate che il loop riportato sopra generi un'attesa di 20 minuti se il clock lavora a 4 MHz.

(b) Quanto dura l'impulso generato dalla sequenza:

```
LDK R0,#1; OUTB RL0,FURN; OUTB RH0,FURN
```

(c) Scrivete una sequenza che generi un impulso di 20 μs.

Nell'esempio precedente il solo criterio da soddisfare era di creare un impulso della durata di venti minuti. Un problema più difficile da risolvere è il mettere insieme il ritardo e le istruzioni in modo che l'intervallo di tempo globale, come anche i suoi sottointervalli, siano di durata misurabile con precisione. Per esempio, se il nostro forno deve rimanere acceso venti minuti una volta ogni ora, allora dovremo fare seguire al nostro impulso di venti minuti un ritardo di quaranta minuti fino al prossimo impulso. Le istruzioni di accensione e spegnimento richiedono un tempo molto breve se confrontato alla scala oraria dei tempi, per cui non influenzano nulla, ma se esse richiedessero un minuto per accendere il forno ed uno per spegnerlo, allora tutto il processo richiederebbe sessantadue minuti e alla fine della giornata il forno si accenderebbe con quarantotto minuti di ritardo.

Naturalmente, noi potremmo ridurre il nostro ritardo da quaranta minuti a trentotto minuti e il problema sarebbe risolto. Ma supponete che ogni tre ore occorra accendere il forno per altri trenta secondi e che ogni dodici ore si debba eseguire un controllo della durata di quindici minuti con successiva analisi dei risultati da parte del calcolatore.

Naturalmente, non importa se o no il vostro forno si accende esattamente allo scoccare dell'ora, ma ci sono molte applicazioni dei calcolatori

dove è importante la precisione temporale. Per esempio, la trasmissione seriale sincrona di bit di dati.

Trasmissione Seriale Sincrona di Bit di Dati

Si immagini un dispositivo che trasmetta al calcolatore dei caratteri ASCII di B-bit ad una velocità di 125 caratteri (1000 bit) per secondo. Supponete che questo avvenga ponendo un registro ad uno o zero per la durata di un millisecondo per ciascun bit di ciascun carattere. Supponete anche che, quando la CPU Z8000 esegue una istruzione INB RL0,DEVICE, il valore nel registro venga letto nel bit zero di RL0. Poi tutto quello che rimane da fare al calcolatore è leggere il registro ad intervalli di un millisecondo. I bit devono essere accumulati sotto forma di byte, ed ogni volta che viene formato un byte, questo deve essere memorizzato da qualche parte. Giusto per stare dalla parte del sicuro, assumiamo di iniziare la lettura a metà strada nell'intervallo di un millisecondo del primo bit; poi leggendo ad intervalli successivi di un millisecondo ci troveremo nel mezzo di ciascun bit con uno spazio su ogni lato pari a 500 μs. Ma se la vostra temporizzazione di un millisecondo viene fermata per soli 250 nanosecondi (un ciclo di clock), allora in 250 caratteri (2 secondi), ci saremo spostati dal centro al bordo dello spazio riservato al bit; saremo perciò fuori sincronismo.

ESERCIZIO 2: (a) Scrivete un segmento di codice che esegua un ritardo di 400 cicli di clock.

(b) Potete sempre scrivere un programma che esegua un ritardo di un qualsiasi numero di cicli di clock?

(c) Cosa succede (per esempio: un terzo di secondo) per i tempi che non sono multipli interi di un ciclo di clock?

I problemi visti nei precedenti esempi sono facilmente risolti introducendo prima e dopo i bit significativi del dato dei bit per risincronizzazione. Se, per esempio, si sa che un determinato bit verrà posto ad uno e che il bit seguente sarà zero, allora tutto quello che dobbiamo fare è iniziare il loop

X: INB RL0,DEVICE; TESTB RL0; JR NZ,X

da qualche parte durante il periodo del bit noto come bit ad uno. Quando si ferma l'esecuzione del loop, ci troveremo all'inizio del bit noto come zero; allora un ritardo pari al tempo (500 μs in questo caso) di un semi-bit ci risincronizzerà nel mezzo del tempo dedicato ad un bit per il bit conosciuto come zero.

Molti dispositivi che trasmettono caratteri ASCII con il protocollo EIA RS-232 usano uno schema simile a questo sulla base del carattere-per-carattere; ciascun carattere ha un bit di start (inizio) ed uno o due bit di stop (fine). Generalmente, il bit di stop ha lo stesso valore che il dispositivo trasmette quando non sta inviando caratteri significativi. Il bit di start ha valore opposto a quello di stop. Dal momento che un dispositivo, non in trasmissione, invia generalmente degli uni per far sapere che è ancora presente, significa che i bit di stop sono degli uni mentre il bit di start è uno zero.

La telescrivente è il terminale per calcolatore più popolare fin dal 1960 (ancora largamente usata). Usa un protocollo chiamato «20 mA current loop» (anello di corrente a 20 mA). Trasmette un bit di start e due bit di stop, e la velocità di trasmissione è di dieci caratteri per secondo. La Figura 6.1 riporta un algoritmo per la lettura dei caratteri provenienti da telescrivente. La Figura 6.2 illustra il programma che lo implementa. Non sono riportate le subroutine DELHALF per eseguire il ritardo pari ad un semi-bit e DELBIT che esegue un ritardo pari ad un bit completo. Si assume che il nome simbolico dell'indirizzo di I/O dell'ingresso dalla telescrivente sia TTYIN.

ESERCIZIO 3:

(a) Dimostrate che la durata di un bit per la telescrivente è di 9.091 millisecondi.

(b) Scrivete una routine DELHALF che esegua un ritardo di 4.55 millisecondi.

(c) Scrivete DELBIT come due chiamate successive di DELHALF.

(d) Quale effetto hanno sulla temporizzazione queste approssimazioni e la mancanza di conteggio delle altre istruzioni di INCH? A quale punto del periodo di 9.091 millisecondi viene letto nello Z8000 il valore dell'ottavo bit? Se il programma non esegue una risincronizzazione su ciascun carattere, dopo quanto tempo avremo perso il sincronismo?

(e) Si spieghi RRCB RL0; RLCB RH0 ed «eseguito se #1 viene fuori dall'altro estremo».

Una cosa che non fa INCH è «l'eco» dei caratteri da inviare indietro alla telescrivente. Questo può essere fatto inserendo l'istruzione

OUTB TTYOUT,RL0

immediatamente dopo INB RL0,TTYIN (dove TTYOUT è il nome simbolico dell'indirizzo di I/O di uscita verso la telescrivente). Esistono due diversi modi di funzionamento dei terminali: *full duplex* e *half-duplex*. Nel modo half-duplex, il carattere corrispondente al tasto digitato dall'operatore viene automaticamente scritto dalla stampante del terminale e con-

1. Aspetta che si abbassi la linea (indica l'inizio di un bit di «start»). Inizializzate l'accumulatore del carattere.
2. Aspetta fino alla metà del bit di start.
3. Aspetta fino alla metà del prossimo bit.
4. Legge il valore del bit.
5. Trasferisce il bit nell'accumulatore del carattere.
6. Se l'intero carattere non è ancora completato ritorna al punto 3.
7. Memorizza il carattere.
8. Fermati se il carattere era un terminatore.
9. Aspetta fino alla metà del prossimo («stop») bit.
10. Ritorna al punto 1.

Figura 6.1 — Algoritmo per la Lettura di Caratteri Provenienti da TTY

temporaneamente inviato (se c'è) al calcolatore. Nel modo full-duplex il calcolatore deve eseguire l'«eco» del carattere ricevuto verso il terminale. Nel funzionamento full-duplex l'eco può essere una risposta immediata del carattere esattamente digitato, oppure l'eco può essere gestito da un programma completamente diverso e può essere costituito dall'invio di uno o più caratteri diversi dal carattere ricevuto. Per esempio, quando l'operatore di un terminale video digita RUBOUT, il programma può inviare indietro come eco una sequenza di tre caratteri BACKSPACE, SPACE, BACKSPACE (sapete perché?). Oppure quando l'utente digita RETURN, può essere inviata in eco la sequenza RETURN, LINEFEED.

In funzionamento half duplex, o con un eco immediato realizzato meccanicamente, sarà utilizzata normalmente una routine simile ad INCH. Quando si userà una interfaccia *parallela* l'eco verrà eseguito con un programma più sofisticato. Con la routine INCH, tutto il tempo del calcolatore viene impiegato per controllare la temporizzazione della trasmissione della telescrivente.

Prima di procedere evidenziamo un altro punto di Figura 6.2: molte righe del programma hanno due istruzioni separate da un punto e virgola. Se il vostro assemblatore è dotato di questa caratteristica, allora potrete decidere se utilizzarla o no. L'utilizzo di questa prestazione è puramente legato a scelte personali. Coloro che la usano dicono che costituisce un importante mezzo per raggruppare insieme le situazioni e per rendere più chiaro il programma; coloro ai quali non piace dicono che l'allontanarsi dalle colonne di istruzioni e di commenti fissi rende un programma confuso e difficile da leggere.

```
!Subroutine per accettare una stringa di caratteri da TTY.
  CALL INLINE; R1 = indirizzo del buffer;
               RH2 = carattere di terminazione.
  Legge e memorizza i caratteri nel buffer indirizzato da R1 fino ad includere il
carattere di terminazione in RH2. Nessun carattere viene inviato in eco. R1 viene
lasciato come puntatore dell'ultimo carattere memorizzato; R0 è perduto!

INLINE:   CALR INCH                    !Prende un carattere!
          LDB @R1,RH0; INC R1          !Lo memorizza!
          CPB RH0,RH2; RET EQ          !Esce se è un terminatore!
          JR INLINE
INCH:     INB RL0,TTYIN; TESTB RL0     !Aspetta l'abbassamento
                                       della linea!
            JR Z,INCH
          LDB RH0,#1                   !Inizializza l'accumulatore
                                       di carattere!
          CALR DELHALF                 !Aspetta il centro del bit di
                                       stato!
LP:       CALR DELBIT                  !Ritardo pari al tempo di un
                                       bit!
          INB RL0,TTYIN                !Leggi il valore!
          RRCB RL0; RLCB RH0           !Sposta il bit nell'accumula-
                                       tore!
            JR NC,LP                   !Fatto se dall'altra estremità
                                       esce # 1!
          CALR DELBIT                  !Ha trovato il primo bit di
                                       stop!
          RET
```

Figura 6.2 — Subroutine per la Lettura di Caratteri da TTY

Nel prossimo capitolo analizzeremo i dispositivi esterni che aiutano la CPU quando deve concentrare tutta la sua attenzione sul controllo della temporizzazione della trasmissione seriale sincrona dei bit.

Trasmissione Parallela Asincrona dei Dati

Nella trasmissione sincrona, il trasmittente e il ricevente «sincronizzano i loro orologi» come le spie o i ladri di banca; ciascuno esegue la sua parte secondo i tempi concordati precedentemente e lascia fare la stessa

cosa all'altro. La trasmissione asincrona, invece, esegue gli scambi utilizzando degli *indicatori* (flags).

Per fare un esempio si può pensare ad una distribuzione postale lungo una strada di campagna; ciascun cliente tiene una piccola bandiera rossa nella cassetta della posta. Se il cliente deve spedire della posta, mette la bandiera fuori dalla cassetta e la posta da spedire all'interno di essa.

Il postino si ferma ad ogni cassetta con la bandiera, preleva la posta da spedire e pone la bandiera dentro la cassetta. Perciò, la trasmissione di posta da un cliente al Servizio Postale segue un protocollo asincrono.

Gli indicatori nello Z8000, come in ogni altra cosa, devono essere realizzati utilizzando il bus indirizzi/dati. Per esempio, nel programma INCH di Figura 6.2, l'istruzione INB RL0,TTYIN è preceduta da CALR DELBIT. DELBIT invece di realizzare un ritardo temporale potrebbe essere un'attesa per un indicatore:

```
DELBIT: INB RL0,TINSTAT; TESTB RL0; JR PL,DELBIT; RET
```

Questa routine attende che sia posto ad uno il bit 7 dello stato letto all'indirizzo simbolico di I/O TINSTAT. Il dispositivo, rispondendo all'istruzione INB RL0,TTYIN, dovrebbe azzerare il bit 7 dell'indicatore di stato e lo dovrebbe lasciare in quella condizione fino a quando non viene letto il prossimo bit.

In effetti questo tipo di protocollo, mediante indicatore chiamato anche *handshaking* (segnale di controllo del trasferimento), non è usato frequentemente nella trasmissione seriale di bit. Tuttavia, questo è il protocollo principalmente usato nella trasmissione *parallela*: trasmissione in cui i vari bit vengono trasmessi contemporaneamente. Una interfaccia parallela, per una telescrivente, invia tipicamente un carattere ASCII di 8-bit ogni volta che viene eseguita la INB RL0,TTYIN. Una routine per l'attesa dell'indicatore, simile a quella riportata sopra, attenderà fino a quando è pronto un intero carattere. Nella Figura 6.3 viene riportata la versione di INCH per la trasmissione parallela.

```
!Versione di INCH per la trasmissione parallela!

INCH:   CALR WAIT                    !Aspetta l'indicatore di TTY!
        INB RL0, TTYIN
        RET

WAIT:   INB RL0,TINSTAT; TESTB RL0; JR PL,WAIT; RET
```

Figura 6.3 — Versione di INCH per Trasmissione Parallela

```
!Routine per l'invio di un carattere in uscita a TTY
  CAAL OUTCH; RL0 = carattere da inviare in uscita!

OUTCH:  PUSH @SR,R0          !Salva R0!
        CALR WAITOUT         !Aspetta fino a quando la TTY è pronta
                             per ricevere!
        OUTB TTYOUT,RL0      !Invia in uscita il carattere!
        POP R0,@SR           !Ricostituisci R0!
        RET
WAITOUT:  INB RH0,TOUTSTAT; TESTB RH0; JR PL,WAITOUT; RET
```

Figura 6.4 — Versione dell'Uscita Equivalente ad INCH

L'uscita parallela asincrona è simile all'ingresso. La Figura 6.4 riporta una routine analoga a INCH per l'invio di un carattere in uscita. È importante notare l'«inizializzazione» del dispositivo. Per certe condizioni, e determinati tipi di interfaccia, è possibile che la prima volta che viene chiamata la routine lo stato di pronto in TOUTSTAT non venga posto ad uno, ne consegue che il programma WAITOUT non eseguirà mai un ritorno al programma chiamante. Siccome supponiamo (come sempre) che OUTB TTYOUT,RL0 posizioni l'indicatore ad uno, per evitare quanto detto prima, occorre scambiare l'ordine di OUTB TTYOUT,RL0 e CALR WAITOUT. Questo causa un'attesa di un decimo di secondo (per una telescrivente sprovvista di buffer), mentre operando nell'altro ordine si esegue per prima tutta l'elaborazione necessaria e si riduce al minimo indispensabile il tempo d'attesa. Un'altra possibile soluzione (quella usuale) consiste nell'avere una routine di inizializzazione che invii all'accensione un carattere «null» alla TTY (o altro dispositivo simile). Più avanti in questo capitolo verrà descritta una soluzione migliore che utilizza le interruzioni.

Larghezza della banda di trasmissione

La massima velocità di trasmissione, detta anche larghezza di banda, costituisce una importante misura della potenza di un calcolatore. La Figura 6.5 illustra due routine di ingresso per alta velocità. Una controlla lo stato, l'altra suppone che il dispositivo sia sempre pronto ed utilizza l'istruzione I/O in versione a blocco con l'opzione della ripetizione. Ciascuna routine viene analizzata per la massima velocità di trasferimento (supponendo che il dispositivo sia sempre pronto). Questa analisi dimostra che la routine di controllo dello stato può supportare frequenze di dati su-

```
!Subroutine per il trasferimento di un blocco di dati in memoria.

   CALL FASTIN;   R0 = n (numeri di caratteri in ingresso)
                  R1 = indirizzo del buffer
                  R2 = indirizzo di I/O dell'ingresso
                  R3 = indirizzo dello stato di I/O

   CALL FASTER;   R0, R1, R2 come sopra
                  R3 non è usato
!
FASTIN:     INB RL4,@R3; TESTB RL4   !Aspetta al massimo 23 cicli!
              JR PL,FASTIN           !Prendi e memorizza il byte
                                                        21 cicli!
            INIB @R1,@R2,R0          !V significa «done»  6 cicli!
              JR NOV,FASTIN          !Totale             50 cicli!
            RET
FASTER:     INIRB @R1,@R2,R0         !Totale      11 + 10n cicli!
            RET
!Supponete che n = 256 (è importante solo per FASTER) la velocità di trasferi-
mento con un ciclo temporale di 250 nanosecondi sono:
    FASTIN:          (50×250×10⁻⁹)⁻¹ =   80,000  byte/secondo
    FASTER:  ((10 + 11/n)×250×10⁻⁹)⁻¹ =  398,289  byte/secondo
!
```

Figura 6.5 — Velocità di Trasferimento parallelo dello Z8000

periori a 80 Kbyte/sec; l'istruzione di I/O per blocco può raggiungere i 398 Kbyte/sec. I dispositivi che trasmettono le parole di 16-bit possono raggiungere una velocità di trasferimento (transfer rate) doppia).

ESERCIZIO 4: Eseguite l'analisi della Figura 6.5 e verificate le velocità di trasferimento ivi date. Utilizzando le istruzioni di caricamento ordinarie o blocco, realizzate una simile analisi per i trasferimenti memoria-memoria.

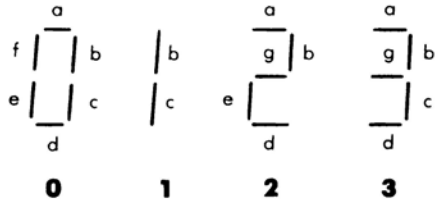
Esempio di Visualizzatore a Sette Segmenti

Molti sistemi a microcalcolatore sono dotati di un dispositivo visualizzatore a sette-segmenti del tipo a LED. La Figura 6.6 riporta l'uso di questo codice a sette segmenti.

ESERCIZIO 5: (a) Verificate la Tabella di Figura 6.6.
(b) Il codice a sette segmenti codifica esattamente tanti caratteri quanti l'ASCII. Potrebbe essere utile? Riferendovi alla Figura 1.15 disegnate le rappresentazioni a sette segmenti che risulterebbero dai codici ASCII di 0, 1, 2, ... A quali caratteri ASCII corrispondono i codici di Figura 6.6?

La Figura 6.7 riporta un programma che cicla su una stringa di cifre esadecimale e le invia in uscita ai LED. Cicla continuamente sui LED, accendendo ognuno di questi per un istante. Ciclando in modo sufficientemente veloce, il programma può tenerli tutti accesi senza effetti di tremolio. La costante k determina per quanto tempo il programma tiene acceso ciascun LED. Se k è troppo piccolo, i LED non appariranno accesi; se è troppo grande, quando viene acceso un LED gli altri si spegneranno.

I sette segmenti coi rispettivi nomi



Rappresentazione di alcune cifre in un codice sette segmenti

0	g	f	e	d	c	b	a

Codifica binaria di una qualsiasi combinazione di segmenti di un byte

Cifra	Codici	Cifra	Codici	Cifra	Codici	Cifra	Codici
0	3F	4	66	8	7F	C	39
1	6	5	6D	9	67	D	5E
2	5B	6	7D	A	77	E	79
3	4F	7	7	B	7C	F	71

Codifica in codice a sette segmenti di tutte le cifre esadecimali

Figura 6.6 — Codice a Sette Segmenti

ESERCIZIO 6: Supponete che il valore di k uguale a 50 sia il minimo necessario per accendere un LED e che i valori più alti siano migliori. Supponete anche che il loop in X debba essere eseguito, minimo, una volta ogni 100 millisecondi per ciascun LED. Qual è il massimo valore possibile di n? Qual è la relazione tra n e k? Naturalmente i valori reali dipenderanno dalle caratteristiche dei LED.

Nel programma di Figura 6.7 ci sono varie altre cose che vale la pena di notare. Per prima cosa si noti il segno «!» posto direttamente sopra alla prima linea di istruzioni. Siccome «!» è la *parentesi* dei commenti all'inizio del programma, lasciandolo sopra come in questo caso, ci aiuta a vedere chiaramente dove è la zona in parentesi e ci garantisce che non abbiamo dimenticato i delimitatori. È solamente una questione di stile.

Occorre notare che questa «subroutine» non esegue mai il ritorno al programma chiamante. Siccome il calcolatore normalmente ha altre cose da fare si potrebbe porre rimedio a questo inconveniente cambiando nell'ultima linea JR GE,DISLED con

```
RET GE
```

quindi il programma chiamante eseguirebbe una sequenza del tipo:

```
DISP:   CALL DISLED        !Una passata sui LED!
        (fai qualcos'altro)
        JR DISP
```

Il tempo utilizzato nella fase «fai qualcos'altro» dovrebbe essere sufficientemente breve per assicurare una visualizzazione priva di disturbi (tremolio).

La TABLE posta in fondo alla Figura 6.7 evidenzia alcuni punti da discutere. Essa deve trovarsi nella memoria *dati* perchè è indirizzata dall'istruzione:

```
LDB RL3,TABLE(R6)
```

Naturalmente, questa tavola di conversione, da esadecimale a sette segmenti, fa parte del programma; se aveste scelto questa soluzione avreste desiderato che la tabella risiedesse in ROM insieme al vostro programma. Per cui in una situazione di questo tipo è più utile non avere la separazione degli spazi degli indirizzi programmi e dati. Dal vostro assemblatore dipende il modo con cui vengono realmente immessi i valori nella TABLE; idealmente vi permetterà di fare qualche cosa di questo tipo:

```
TABLE:  HEXBYTES (3F, 6, 5B, 4F, 66, 6D, 7D, 7, 7F,
                  67, 77, 7C, 39, 5E, 79, 71)
```

```
!Programma di visualizzazione con LED (versione non segmentata)
  CALL DISLED; R0 = n (numero di LED)
               R1 = indirizzo di n cifre esadecimali, un/byte
               R2 = indirizzo di I/O del primo LED (gli altri sono gli in-
               dirizzi successivi)

  Uso dei registri interni
    RL3     =  codice corrente a sette segmenti da inviare in uscita
    R4      =  contatore LED (i valori ciclano attraverso 0,1,....N –
               1,0,1,....)
    R5      =  puntatore dell'indirizzo di I/O (contiene sempre R2 + R4)
    R6      =  valore esadecimale della cifra corrente (dalla stringa indi-
               rizzata da R1)
    R7      =  contatore di ripetizione per «verifica» dei LED!
!
DISLED:  CLR R4; LD R5,R2        !Inizializza il contatore ed il punta-
                                 tore I/O!
         CLRB RH6                !Cambia solamente il byte meno si-
                                 gnificativo di R6!
LOOP:    LDB RL6,R1(R4)          !Cifra esadecimale successiva!
         LDB RL3,TABLE(R6)       !Codice a sette segmenti della cifra
                                 successiva!
         LD R7,#k                !Contatore di «verifica»!
LIGHT:   OUTB @R5,RL3            !Accende la cifra!
         DEC R7; JR GT,LIGHT     !Si assicura che si veda!
         INC R4; INC R5          !Cifra successiva, LED successivo!
         CP R4,R0; JR GE,DISLED  !Se ctr ≤n allora salta a DISLED!
         JR LOOP                 !        altrimenti a LOOP!
TABLE:   (16-byte corrispondenti alla Figura 6.6)
```

Figura 6.7 — Programma per la Visualizzazione Mediante LED

Mentre è possibile che voi dobbiate fare qualche cosa di questo tipo:

TABLE: bval %3F; bval %6; bval %5B; ecc.

Catalogatore di I/O

Esistono due modi principali per catalogare la trasmissione dati da e verso il calcolatore: *polling* ed *interruzioni.*

Il postino che gira tutto il giorno cercando le bandierine sulle cassette postali lavora secondo uno schema di tipo polling. L'impiegato postale che, sospende l'attività standard ogni volta che viene infilata una lettera attraverso la buca delle lettere posta nella sala postale e poi riprende l'attività standard dopo avere sistemato la lettera, utilizza un sistema ad interruzione per catalogare la trasmissione.

Nel calcolatore, il polling esegue ripetitivamente una sequenza del tipo di quella riportata in Figura 6.8. Lo stato di ogni dispositivo viene controllato con una sequenza di codice di questo tipo

FASTIN: INB RL4,@R3; TESTB RL4; JR PL,FASTIN

presa da Figura 6.5, ma invece di JR PL,FASTIN, che attende fino a quando il dispositivo è pronto, la sequenza di polling salterà al test successivo — evitando la chiamata della routine di servizio del dispositivo.

Se oltre a servire i dispositivi il microprocessore non ha niente altro da fare questa attività viene integrata nel loop di polling. Per esempio, la chiamata della subroutine potrebbe avvenire poco prima di

«ritorna indietro a LOOP»

di Figura 6.8. Il polling diviene più complicato se c'è una urgenza oppure una limitazione di tempo (esempio i LED devono essere serviti ad ogni decimo di secondo). La CPU potrebbe gestire questi problemi tenendo conto delle *priorità*: un modo semplice consiste nel disporre i controlli dei dispositivi di Figura 6.8 in ordine di importanza. Poi, dopo ogni chiamata

LOOP: se deve essere servito il Dispositivo 1, chiama la routine di servizio del Dispositivo 1
Se deve essere servito il Dispositivo 2, chiama la routine di servizio del Dispositivo 2

•
•
•

Se deve essere servito il Dispositivo N, chiama la routine di servizio del Dispositivo N
Torna a **LOOP**

Figura 6.8 — Tipica Sequenza di Polling

della routine di servizio del dispositivo, il programma invece di eseguire il test successivo dovrebbe saltare a LOOP.

La figura 6.8 rappresenta quello che viene chiamato un «*round robin*» (lista di circolazione); il salto a LOOP, dopo ciascuna chiamata della routine di servizio del dispositivo lo trasforma in uno schema a *priorità fissa*. Nello schema a priorità fissa *non* vengono fissate le priorità.

Un dispositivo a bassa priorità che non è stato servito in un certo momento può diventare in un certo periodo di tempo, un dispositivo ad alta priorità. Si potrebbe risolvere il problema variando le priorità in funzione della situazione. In questo settore sono stati fatti dei lavori piuttosto sofisticati, ma il vantaggio ottenuto per i sistemi ordinari non è pari allo sforzo compiuto. Nella maggior parte dei casi è meglio utilizzare un round robin, che ricorra a routine di servizio scritte appositamente ed eseguite ogni volta che vengono chiamate, per un tempo il più breve possibile.

Se il calcolatore oltre a servire i dispositivi ha molte altre cose da fare, allora diviene difficile eseguire un polling veloce. Fortunatamente lo Z8000 fornisce, per mezzo delle interruzioni, un valido supporto per eseguire la catalogazione. Nel Capitolo II abbiamo parlato dell'area di stato del programma (PSA) e del salvataggio e ripristino dello stato della CPU quando si ha una interruzione. Nel Capitolo IX vedremo come predisporre il PSA. Nella discussione che segue abbiamo solamente bisogno di sapere che quando si verifica una interruzione la CPU fornisce un qualche mezzo per trasferire il controllo ad una opportuna routine di servizio.

Le routine di servizio del dispositivo sono essenzialmente le stesse indipendentemente dal fatto che vengano catalogate con il sistema polling o interruzione. In effetti, il tipo più semplice di interruzione (la NVI nello Z8000) è gestita da una sequenza analoga a quella di Figura 6.8. La differenza consiste nel fatto che con l'interruzione il programma arriva solamente a LOOP quando qualche dispositivo ha segnalato di avere bisogno di essere servito. Si può risolvere il problema eseguendo tutto il lopp e facendo una IRET alla fine; oppure ciascuna routine di servizio del dispositivo può contenere una IRET. Nell'ultimo caso se deve essere servito un altro dispositivo si verifica una interruzione ed il programma torna a LOOP.

Le limitazioni del loop di Figura 6.8 vengono completamente eliminate usando l'interruzione vettoriale. Ciascun dispositivo collegato a VI deve fornire un identificatore alla CPU. La CPU è in grado di trovare la locazione di memoria che contiene l'indirizzo della routine di servizio del dispositivo e trasferire il controllo direttamente alla routine di servizio. Una locazione di memoria dedicata a contenere l'indirizzo della routine di ser-

vizio da utilizzare in questo modo è chiamata *vettore di interruzione* (dall'equivalente latino di freccia-vector). Una routine di servizio raggiunta in questo modo deve finire con una IRET perchè può eseguire solo questo tipo di ritorno.

Le routine di servizio del dispositivo possono essere molto semplici o complesse specialmente se il dispositivo può essere coinvolto in diversi tipi di attività. La complessità è dovuta al fatto che si devono ricostruire i contesti delle routine ogni volta che sono chiamate; perciò, devono ricordarsi che cosa stavano facendo e dove esattamente si trovavano all'interno della routine.

Un Esempio

È difficile padroneggiare questo tipo di programmazione senza avere visto un esempio valido. Per questa ragione il resto di questo capitolo sarà dedicato alla presentazione di routine di servizio di dispositivo necessarie per gestire un terminale (esempio, CRT, telescrivente, ecc.).

Un terminale è costituito da due parti: una tastiera ed una «stampante». La routine di servizio per la tastiera è molto semplice siccome tutto quello che deve fare è accettare caratteri. La routine di servizio della stampante è più complicata, siccome può essere usata sia per inviare messaggi in uscita all'operatore sia per inviare indietro in eco i caratteri accettati dalla routine di tastiera. L'eco è ulteriormente complicato perchè alcuni caratteri della tastiera possono generare stringhe di uscita di lunghezza maggiore di un carattere (esempio, RUBOUT può generare «BACKSPACE, SPACE, BACKSPACE» per un video onde cancellare il carattere precedente e riposizionare il cursore dove è stato cancellato il carattere); quando la routine di servizio termina la stampa di una stringa ha bisogno di conoscere se ha finito di lavorare o se deve ritornare ad eseguire un eco.

La Figura 6.9 mostra un algoritmo per la routine di servizio dell'uscita verso terminale — la routine che invia realmente i caratteri alla TTY o al video ogniqualvolta il dispositivo interrompe per dire che è pronto per un altro carattere. Viene presentato in «pseudocodice»; questa è una forma alternativa per presentare un algoritmo ricorrendo alle strutture di controllo che si trovano nei linguaggi ad alto livello. Questa tecnica può essere molto più chiara dell'altra, che abbiamo discusso per gli algoritmi in un caso simile a questo, in cui c'era ovunque una grande quantità di salti.

Il concetto che sta dietro lo pseudocodice è il seguente: cercare di esprimere l'algoritmo chiaramente, mescolando descrizioni verbali con istruzioni o strutture di controllo in una approssimazione casuale di un qualche linguaggio di programmazione che vi sia famigliare e di facile comprensione ad una qualsiasi persona che debba leggerlo.

Routine di Servizio d'Uscita per Terminale:

Se FLAG = «ingresso» allora salta ad **Eco**
FLAG = «eco della stringa» allora salta ad **Eco della Stringa**
FLAG = «uscita» allora salta alla **Stampa**

Altrimenti salta ad **Errore!** Non dovrebbe entrare nella routine se è nella condizione «fatto» o «attesa»!

Eco: Preleva il carattere successivo dal buffer di ingresso
Se il buffer è vuoto, poni FLAG = «attesa» e IRET
Altrimenti se il carattere = RUBOUT o BS, allora salta a **Rubbing**

il carattere = ESC, allora salta ad **Escape**
il carattere = CR, allora salta alla **Terminazione**

Altrimenti salta alla **Stampa del Carattere!** Se non è un carattere speciale, invialo in eco!

Rubbing: Rinnovi l'ultimo carattere di LINE
Se non c'è nulla in LINE, allora salta ad **Eco**
Altrimenti (preleva l'ultimo carattere posto in LINE; poni PTS a «B S, SPACE, B S»; poni FLAG = «eco della stringa»; salta a **Eco della Stringa**).

Escape: Aggiungi il carattere ESC a LINE; poni PTR a «$/»
poni FLAG = «eco della stringa»; salta a **Eco della Stringa.**

Terminazione: Aggiungi un NUL a LINE; poni PTR a «CR, LF»
posiziona FLAG = «uscita»; salta a **Stampa**

Stampa del Carattere: Aggiungi il carattere a LINE; invialo in uscita; IRET

Eco della Stringa: Se PTR punta a «NUL», allora (poni FLAG = «ingresso»; salta a **Eco**)

Altrimenti (invia in uscita il carattere puntato da PTR; INC PTR; IRET)

Stampa: Se PTR punta a «NUL», allora (poni FLAG = «fatto», IRET)

Altrimenti (invia in uscita il carattere puntato da PTR; INC PTR; IRET)

Errore: (da specificare)

Figura 6.9 — Pseudocodice per la Routine di Servizio dell'Uscita per Terminale

ESERCIZIO 7: Cercate di scrivere l'algoritmo di Figura 6.9 nel formato inizialmente usato per gli algoritmi. Non preoccupatevi di finirlo; andate avanti quanto basta per convincervi che il risultato sarebbe veramente difficile da comprendere.

La routine di servizio dell'uscita del terminale è giusto una parte di un insieme di programmi. Nel Capitolo VIII presenteremo un programma di inizializzazione che viene chiamato ogni volta che il programmatore deve inviare qualcosa in uscita alla stampante/video o ricevere in ingresso dalla tastiera. Il programma di inizializzazione aspetta fino a quando è terminato il programma precedente, per predisporre i propri valori nelle variabili di controllo FLAG e PTR ed inizializza LINE. (Guardate oltre alla Figura 8.1).

Come vedremo quando analizzeremo l'algoritmo di Figura 6.9, FLAG ci informa sul lavoro della routine di servizio, PTR punta al prossimo carattere che deve essere inviato in uscita e LINE forma in ingresso la linea che viene digitata dall'operatore. Questi e gli indirizzi di I/O e il buffer di ingresso usati costituiscono il contesto in cui operano le routine di servizio per l'uscita verso e l'ingresso da terminale. Il programma inizializza il contesto per le routine di servizio di ingresso e uscita.

Per ultimo, la routine di inizializzazione deve far partire l'intero processo, siccome si sarà già avuta l'interruzione per il «ready» (pronto) dalla stampante/video; la parte circuitale lo avrà riconosciuto e quindi non si potrà riavere nuovamente fino a quando non viene inviato in uscita un altro carattere. La routine di inizializzazione segnalerà l'interruzione mediante una funzione SC. I dettagli saranno discussi nel Capitolo IX.

Con questa idea generale di ciò che fa la routine di inizializzazione, analizziamo la routine di servizio di uscita per terminale di Figura 6.9. Durante l'analisi capiremo che cosa deve fare la routine di ingresso. Dopo che abbiamo codificato le routine di ingresso e uscita, scriveremo la vera routine di inizializzazione, riportata nel Capitolo VIII.

Per prima cosa, la routine guarda FLAG per capire che cosa sta facendo. FLAG può avere cinque valori: *ingresso, eco della stringa, uscita, attesa e fatto.* Questi hanno i seguenti significati:

ingresso (input): Si sta costruendo una linea in ingresso; dovrebbe essere processato il prossimo carattere dal buffer di ingresso.

eco della stringa (echostring): L'ultimo carattere di ingresso processato richiede che venga inviata in eco una stringa speciale di caratteri; il carattere successivo di questa stringa deve essere inviato in uscita.

uscita (output): Una stringa di testo sta per essere inviata in uscita alla stampante/video; il carattere successivo a questa stringa dovrebbe essere inviato in uscita.

attesa (sleep): All'ultima interruzione, FLAG aveva il valore «ingresso», ma nel buffer di ingresso non c'erano caratteri. La prossima volta che entra un carattere la routine di ingresso dovrebbe riposizionare FLAG su «ingresso» e trasferirlo a questa routine di attesa.

fatto (done): All'ultima interruzione, FLAG aveva il valore «uscita» ma PTR stava puntando al carattere di terminazione della stringa in uscita; non dovrebbe verificarsi niente altro fino a quando la routine di inizializzazione non inizializza qualche cosa altro.

Una volta che si è deciso che cosa deve fare il programma si passa alla parte relativa al codice. Le parti *Eco della Stringa* per FLAG = «eco della stringa» e *Stampa* per FLAG = «uscita» sono semplici e molto simili tra di loro; ciascuna invia in uscita il carattere successivo ed incrementa PTR. Esse differiscono nell'esecuzione della fine della stringa: la *stampa* pone FLAG = «fatto»; l'*eco della stringa* riposiziona FLAG = «ingresso» e va a prendere il carattere successivo in ingresso. *L'eco* è la parte più complicata. Preleva il carattere successivo dal buffer di ingresso e lo processa in uno dei possibili quattro modi in funzione del tipo di carattere. Se il buffer di ingresso è vuoto, come detto sopra, FLAG viene posto in «attesa».

La stampa del carattere è il caso ordinario: se non c'è niente di speciale per quanto riguarda il carattere, questo viene inviato indietro in eco all'operatore ed aggiunto nel buffer della linea di ingresso LINE.

La terminazione, viene eseguita quando l'operatore digita RETURN: viene terminato LINE per cui il programmatore, che aveva chiamato la routine di inizializzazione, conoscerà il numero di caratteri digitati; viene inviata come eco la sequenza RETURN, LINEFEED per portare il carrello/cursore nella posizione all'estremo sinistro della linea successiva; siccome l'ingresso è stato eseguito FLAG viene posizionato su «uscita».

La cancellazione viene eseguita quando l'operatore digita RUBOUT oppure BACKSPACE: viene tolto da LINE l'ultimo carattere precedente ai due sopra menzionati; il carattere viene cancellato dal video mediante la sequenza BACKSPACE, SPACE, BACKSPACE; poi si ritorna in ingresso.

Il carattere *ESCAPE* viene riportato come esempio di un prototipo di molti caratteri speciali che potreste voler gestire in modi particolari. Si noti che ESC viene aggiunto a LINE, ma all'operatore vengono inviate indietro in eco i due caratteri «**$/**».

ESERCIZIO 8: (a) C'è un «errore» connesso con ESC; la sequenza ESC, RUBOUT non agisce correttamente su LINE ma lascia il carattere «**$**» sul video. Pensate ad una soluzione?

(b) Quali altri caratteri hanno bisogno di un trattamento speciale, e quali problemi pongono? Che cosa succede con TAB, LINEFEED o altri caratteri di «controllo»?

La Figura 6.10 riporta la routine di servizio dell'ingresso del terminale usando la stessa forma di pseudocodice utilizzato per la routine di uscita. Adesso possiamo scrivere il codice reale di questa routine, ma prima di questo dobbiamo studiare i dettagli del *context switching* (commutazione del contesto).

Abbiamo detto che la routine di inizializzazione, inizializza il contesto delle routine di servizio e nella versione in pseudocodice abbiamo indicato come la routine di servizio debbono accedere a questo contesto. Facciamo in modo che questo venga realizzato mediante un blocco di memoria che inizia all'indirizzo simbolico CONBLK; questo blocco contiene tutti gli elementi del contesto utili per le routine di servizio. Quando vengono chiamate le routine di servizio, i registri verranno salvati nello stack e il blocco del contesto viene ripristinato nei registri.

Routine di Servizio per l'Ingresso da Terminale:

```
Prendi il carattere dalla tastiera
Mettilo nel buffer di ingresso
Se FLAG = «sleep» allora salta a echoing oppure IRET
```

Figura 6.10 — Pseudocodice della Routine di Servizio di Ingresso da Terminale

Quando terminiamo, verranno ripristinati i registri appena prima di eseguire la IRET. La Figura 6.11 riporta il codice di due routine che gestiscono quanto detto prima: TYENT e TYEXIT. Ciascuna delle routine di servizio inizia con una chiamata a TYENT, per predisporre il contesto, e termina con una chiamata a TYEXIT, per ripristinare i registri prima di eseguire la IRET.

Occorre discutere molti elementi della Figura 6.11. Per prima cosa si noti la definizione dei simboli (esempio TYCON = R2, FLAG = RH2). La loro forma dipende dal particolare assemblatore Z8000 usato, qualsiasi assemblatore dovrebbe essere in grado, al limite, di fornire questa possibilità di definire molti simboli. Forse il vostro assemblatore avrà altre ca-

ratteristiche che renderanno più facili e meno soggetti ad errori i futuri cambiamenti. Costruzioni del tipo:

PTR = TYCON + 1

RING = TYCON + 2

o anche

NTYCX = TYRT − TYCON + 1

rendono impossibile il cambiamento di una delle coppie di una relazione simbolica senza che venga cambiata l'altra.

Occorre fare attenzione a queste definizioni perché queste devono essere usate non solo in TYENT e TYEXIT ma anche nelle nostre routine di servizio TYOUT e TYIN e nella routine di inizializzazione TYIO.

Un punto essenziale della Figura 6.11 è che la maggior parte del contesto è costituita da *indirizzi*; come viene riportato di seguito, questi programmi funzionano solamente con lo Z8000 in versione *non-segmentata*. Le versioni segmentate sono analoghe, ma diverse.

La codifica di TYENT e TYEXIT è molto breve, ma non semplice. La prima complicazione deriva dai programmi chiamati come subroutine, questo richiede che il ritorno sia salvato nello stack mentre in realtà nello stack sono salvati i valori dei registri. Siccome dobbiamo rimuovere il ritorno, memorizzato dove abbiamo salvato i registri, facciamo un atto generoso ed aggiungiamo l'elemento TYRT al blocco di contesto; nessuna routine utilizza TYRT, ma la sua presenza può servire quando ne avremo bisogno per il debug (correzione).

Queste sono le operazioni con cui TYENT e TYEXIT si scambiano il contesto:

(1) TYENT scambia il valore corrente di R8 con il valore di ritorno salvato nello stack poi, utilizzando la stessa tecnica descritta nella Figura 3.9, salva R0-R7.

(2) Poi TYENT carica in memoria R2-R7 del blocco di contesto, lasciando inalterato R8.

(3) Infine, TYENT ritorna al chiamante usando il ritorno salvato in R8.

(4) TYEXIT inizia prelevando l'indirizzo di ritorno da R8.

(5) Poi TYEXIT salva R2-R8 nel blocco di contesto della memoria.

(6) Infine TYEXIT ricostituisce i valori dei registri R0-R8 (invertendo il punto 1 sopra riportato) e ritorna al programma chiamante.

```
!Commutazione del Contesto di I/O per Terminale (versione non-segmentata)

   CALL TYENT – posiziona il contesto; lascia C inalterato
   CALL TYEXIT – ricostituisce il contesto; lascia C inalterato

Uso dei registri:
   R0,R1 sono registri scratch!

! TYCON = R2     !Blocco del contesto – come CONBLK in memoria!
   FLAG = RH2    !Dice che cosa sta facendo TYOUT **MUST BE FIRST**!
   CHAR = RL2    !Carattere corrente!
   PTR  = R3     !Indirizzo del successivo carattere che TYOUT deve inviare
                 fuori!
   RING = R4     !Indirizzo del buffer di ingresso della tastiera!
   LINE = R5     !Indirizzo della LINE di ingresso in preparazione!
   IOIN = R6     !Ingresso e!
   IOUT = R7     !Indirizzi degli I/O di uscita!
   TYRT = R8     !Indirizzo dell'ultimo chiamante (è l'ultimo registro salvato)!
  NTYCX = 7              !Numero dei registri del contesto!
  TYCXB = 2*NTYCX        !Ciascuno occupa 2 byte di memoria!
  NTYRG = NTYCX + 2      !Totale del conteggio dei registri di scratch!
  TYRGB = 2*NTYRG

TYENT:  EX TYRT,@SR                          !Scambia i ritorni dello stack!
        DEC SR,#TYRGB – 2                    !Salva gli altri registri!
        LDM @SR,R0,#NTYRG – 1
        LDM TYCON,CONBLK,#NTYCX – 1          !Carica il contesto
                                             (escluso TYRT)!
        JP @TYRT                             !Ritorna al chiamante!

TYEXIT: POP TYRT,@SR                         !Esegue il ritorno in TYRT!
        LDM CONBLK,TYCON,#NTYCX              !Aggiorna il contesto!
        LDM R0,@SR,#NTYRG – 1                !Ricostituisce i registri
                                             (escluso TYRT)!
        INC SR,#TYRGB – 2
        EX TYRT,@SR; RET                     !Ricostituisce TYRT ed esce!
```

Figura 6.11 – Routine di Commutazione del Contesto per l'I/O Terminale

Si noti che il successo di questo processo dipende dal fatto che TYRT è messo per ultimo nei registri e che i registri di scracth (registri di servizio) sono posti per primi. Dobbiamo però ricordare che, se noi aggiungiamo

sempre qualche cosa di più al blocco di contesto, i nuovi elementi devono essere posti da qualche parte dopo R1 e prima di TYRT. Per cui occorre ridefinire NTYCX.

La Figura 6.11 ci permette di scrivere la routine TYOUT corrispondente alla Figura 6.9. Questa viene riportata, suddivisa in due parti, nelle Figure 6.12 e 6.13. Per avere dei nomi relativamente brevi, sono stati fatti alcuni cambiamenti a questi altrimenti le Figure 6.12 e 6.13 sarebbero l'interpretazione diretta della Figura 6.9.

L'istruzione «If Case... go to...» di Figura 6.9 è stata implementata usando la routine TRAN che discuteremo nel Capitolo VIII. Essa prende un valore da R0, lo ricerca in una *tabella associativa* del tipo TYGO o CHRGO e sostituisce R0 con la seconda parte della prima coppia in cui il valore originale di R0 costituisce la prima parte. Per esempio, riferendosi a TYGO, se il valore originale di R0 è ECSTR, allora TRAN lo sostituirà con STREC. Se non viene trovato l'elemento nella tabella, viene fornito il valore di default (valore attribuito automaticamente nel caso in cui non sia stata data un'assegnazione ad una variabile) che segue lo zero. Anche TRAN utilizza C per indicare se l'elemento è stato trovato; questo è simile a ciò che viene fatto in SAYNX, la breve routine riportata in fondo alla Figura 6.12. Questo è un modo utile per ricevere delle informazioni dalle subroutine quando il programma chiamante può o non può prenderne cura (esempio usando TRAN come sopra non guarderemo mai al valore di C che essa restituisce come informazione). Per rendere questo schema utilizzabile in modo speciale, si dovrebbe seguire una determinata convenzione. Quella usata in questo libro è

C = 0: Normale, prevista, ritorno senza errori

C = 1: Anormale, imprevista, ritorno con errori

Per esempio, come vedremo nel Capitolo VIII, se l'elemento viene trovato nella tabella TRAN restituisce C = 0, se non viene trovato C = 1. Per le routine che controllano delle condizioni, l'interpretazione di C dovrebbe dipendere dal nome della routine. Per esempio, se CHKRDY è una routine che controlla se c'è qualche cosa di «libero» o «occupato», allora dovremo usare

C = 0: libero

C = 1: occupato

Seguendo una simile convenzione, avrete qualche cosa in meno di cui

```
!Routine di Servizio di Uscita per Terminale!

TYOUT:  CALR TYENT
        LDB RL0,FLAG; CLRB RH0      !Usa FLAG!
        LDA R1, TYGO                !e TYGO!
        CALL TRAN; JP @R0

!Tabella degli indirizzi di elaborazione dei vari valori di FLAG!

TYGO:   INPUT; ECHO; ECSTR; STREC; OUTPUT; PRINT; 0; ERROR
STREC:  !Eco della stringa!
        CALR SAYNX; JR NC,TYEX      !Invia in uscita il prossimo, esce se
                                    non è finita!
        LDB FLAG,#INPUT; JR ECHO    !Se è finita, ritorna ad eseguire l'e-
                                    co!
PRINT:  !Stampa!
        CALR SAYNX; JR NC,TYEX      !Invia in uscita il prossimo, esce se
                                    non è finita!
        LDB FLAG,#DONE; JR TYEX     !Se è finita, dice «done»!
TYEX:   JR ECEX                     !Finisce l'elaborazione dell'inter-
                                    ruzione!
ERROR:  JR ECEX                     !Non dovrebbe essere qui — igno-
                                    ralo!
SAYNX:  LDB RL0, @PTR; CPB RL0,#TRM
          JR EQ,NOEX                !Preleva il carattere successivo,
                                    comunica se non c'è niente!
        OUTB @IOUT,RL0; INC PTR     !Lo invia in uscita se non è il ter-
                                    minatore!
        RESFLG C; RET               !C = 0 Se non finisce!
NOEX:   SETFLG C; RET               !C = 1 Se finisce!
```

Figura 6.12 — Routine di Servizio di Uscita per il Terminale (Escluso Eco)

preoccuparvi, siccome conoscerete sempre il significato dello stato di ritorno senza dover riguardare il tabulato della routine.

Le stringhe di caratteri CRS, BSS e ESCS in Figura 6.13 non sono probabilmente nella forma riconoscibile da un qualsiasi particolare assemblatore Z8000. Il concetto base è il seguente: i caratteri i cui nomi simbolici sono riportati in parentesi, saranno assemblati in byte di memoria adiacenti con inizio da quello uguale alla label (etichetta) data. Si sup-

```
!Sezione Eco di TYOUT!

ECHO:  LD R1,RING; CALL GETRNG          !Prossimo carattere di in-
                                        gresso!
         JR NC,EC1
       LDB FLAG,#SLEEP; JR ECEX         !Attende se non c'è!
EC1:   LDB CHAR,RL0                     !Salva CHAR!
       CLRB RH0; LDA R1,CHRGO           !Usa CHRGO & CHAR!
       CALL TRAN; JP @ R0
!Tabella degli indirizzi di elaborazione dei vari valori di CHAR!
CHRGO: RUBOUT;ECRUB; BS;ECRUB; CR;ECCR; ESC; ECES; 0; ECC

ECRUB: !Carattere di RUBOUT!
       LD R1,LINE; CALL BACKUP          !Prende indietro l'ultimo ca-
                                        rattere!
         JR C,ECHO                      !Niente da Prendere!
       LDA PTR,BSS; LDB FLAG,#ECSTR     !L'eco elimina i caratteri!
       JR STREC
BSS:   BYTES(BS,SPACE,BS,TRM)

ECCR:  !Carattere di terminazione!
       LD R1,LINE; LDB RL0,#TRM         !Termina LINE!
       CALL ADLINE
       LDA PTR,CRS; LDB FLAG,#OUTPUT !Invia in eco CR,LF!
       JR PRINT
CRS:   BYTES(CR,LF,TRM)

ECES:  !Carattere di ESCape!
       LD R1,LINE; LDB RL0,CHAR         !Aggiunge ESC a LINE!
       CALL ADLINE
       LDA PTR,ESCS; LDB FLAG,#ECSTR !Invia in eco $,/!
       JR STREC
ESCS:  BYTES('$','/',TRM)

ECC:   !Stampa il carattere!
       LD R1,LINE; LDB RL0,CHAR         !Aggiunge CHAR a LINE!
       CALL ADLINE
       OUTB @ IOUT,CHAR                 !Stampa il carattere!
ECEX:  CALR TYEXIT; IRET                !Commuta il contesto &
                                        IRET!
```

Figura 6.13 — Routine di Servizio di Uscita per il Terminale (Parte Eco)

```
!Routine di Servizio per l'Ingresso da Terminale!

TYIN:   CALR TYENT                          !Posiziona il contesto!
        INB CHAR, @ IOIN; CALL PUTRNG       !Preleva e memorizza il carat-
                                            tere!
        CPB FLAG,#SLEEP; IR EQ,ECHO         !Risveglia l'uscita se sta atten-
                                            dendo!
        CALR TYEXIT; IRET                   !Ricostituisce il contesto e
                                            procede!
```

Figura 6.14 — Routine di Servizio per Ingresso da Terminale

pone anche che l'assemblatore aggiunga un byte zero extra per assicurarsi che l'istruzione successiva (esempio, LD R1, LINE dopo BSS) sia assemblata ad un indirizzo pari e che la label successiva (esempio, ECCR:) abbia il valore corretto. Gli assemblatori variano a seconda di come vengono trattati questi argomenti; dovete individuare ed usare le convenzioni del vostro assemblatore.

C'è un'altra cosa da dire a riguardo di queste stringhe e a proposito delle tabelle TYGO e CHRGO: queste vengono usate in modo da essere poste nella memoria dati. Questo dimostra perché non si dovrebbero usare, nella programmazione non specializzata, gli spazi di memoria separati per dati ed istruzioni. Naturalmente, se esse devono trovarsi in spazi separati di memoria, l'assemblatore deve fornire un modo per renderlo possibile, inclusa l'assegnazione dei valori corretti delle label del tipo BSS e TYGO. Il caricatore dovrà anche fornire l'«inizializzazione» della memoria che richiederà una specifica configurazione circuitale adatta all'uso.

Nel Capitolo VIII verranno presentate le routine ADLINE e BACKUP per la gestione di LINE e le routine GETRNG e PUTRNG (viste in Figura 6.14) per la gestione di RING. LINE è semplicemente un insieme di caratteri; RING è un buffer di ingresso first infirst out (primo ad entrare — primo ad uscire). Il suo nome deriva dalla implementazione come buffer ad anello (RING) — il modo più semplice per realizzare un buffer FIFO via software.

Infine, la Figura 6.14, illustra la routine di servizio di ingresso TYIN. Essa segue, molto da vicino, la struttura di Figura 6.10.

Nei programmi di Figura 6.12, 6.13 e 6.14 sono stati usati vari nomi simbolici che non sono stati definiti in modo esplicito: CR, LF, BS, PACE, TRM (= NUL, il carattere di terminazione) e i valori di FLAG, INPUT, ECSTR, OUTPUT, SLEEP, DONE. Per i valori di FLAG andrà

bene un qualsiasi valore distinto nel campo da 1 a 255. Gli altri sono nomi simbolici di caratteri ASCII e devono avere dei valori corrispondenti a quelli di Figura 1.15. Con questo abbiamo finito l'esempio.

ESERCIZIO 9: (a) Si noti che la prima istruzione ad ECRUB, ECCR, ECES, ECC è LD R1,LINE; esiste un qualche modo per evitare questa ripetizione? Che cosa si può fare per LDB RL0,CHAR che appare ad ECC e ECES? E che cosa si può dire per le tre ripetizioni di CALL ADLINE?

(b) Supponete di dover gestire una mezza dozzina o quasi di caratteri nello stesso modo in cui è gestito ESC: mettete il carattere in LINE ed inviate come eco una stringa speciale. Esiste un modo uniforme per fare questo usando TRAN ed un numero maggiore di tabelle?

(c) Fornite la prestazione TAB che memorizzi il carattere TAB in LINE e che invii come eco degli zeri, o più spazi, per spostare la testina di stampa o il cursore nella posizione successiva di una tabella di stop di tabulazione. Si assuma che l'indirizzo della tabella sia un altro elemento del contesto. Fate i cambiamenti necessari per le definizioni del contesto.

ESERCIZIO 10: Scrivete la versione segmentata della routine di gestione dell'interruzione da terminale. Quali cambiamenti potrebbero rendere più simili le due versioni?

ESERCIZIO 11: Nel Capitolo IX scopriremo che esiste la possibilità di prevenire le interruzioni da uno dei dispositivi terminali mentre stiamo gestendo le interruzioni di un altro. Che cosa accadrebbe se l'esecuzione di TYOUT fosse sospesa per permettere che l'ingresso di una interruzione venga processata da TYIN? o Viceversa? Ricordatevi, che ciascuna routine possiede una copia del contesto nei registri, che viene ricopiata in memoria quando viene eseguita. Questo problema sarà discusso nel Capitolo VIII.

Capitolo VII

I componenti Periferici dello Z8000

Lo Z8000 è stato progettato come una famiglia di componenti che condivide una struttura comune di bus e che fornisce tutti i pezzi necessari per supportare le applicazioni tipiche dei calcolatori. Richiede troppo tempo descrivere in questo libro come tutti questi dispositivi siano usati insieme; per questo è stato pianificato un libro di *Applicazione dello Z8000* nelle pubblicazioni future. In questo capitolo daremo una descrizione generale di questa famiglia di componenti e discuteremo le operazioni interne della CPU Z8000 che hanno a che fare con I/O.

Istruzioni Speciali di I/O

Molte volte abbiamo fatto allusioni ad istruzioni «speciali» di I/O concepite per essere usate con l'unità di gestione della memoria. Queste istruzioni non possono accedere alle linee $AD_7 - AD_0$ del bus indirizzi/dati e devono usare le linee $AD_{15} - AD_8$ per tutte le informazioni sugli indirizzi/dati. Ci troviamo ora di fronte ad un punto poco chiaro: nell'*idea originale* le istruzioni speciali di I/O utilizzavano $AD_{15} - AD_8$ per convogliare gli indirizzi e i dati a 16 bit; questo *fu realizzato implementando* le istruzioni speciali di I/O esattamente come se fossero normali istruzioni di I/O, utilizzando quindi $AD_{15} - AD_0$ per le informazioni a 16 bit. L'unica differenza è che le istruzioni speciali di I/O producono un certo valore sulle linee di stato $ST_3 - ST_0$ mentre le normali istruzioni di I/O ne producono un altro. (Vedasi Figura 2.12).

Che cosa succede esattamente quando la CPU Z8000 esegue una istruzione di I/O? Esattamente la stessa cosa che si verifica per un accesso alla memoria: per accedere ad una parola viene inviato in uscita su

$AD_{15} - AD_0$ un indirizzo di 16 bit, poi sulle stesse linee vengono inviati in uscita o ricevuti in ingresso i 16 bit del dato. Per accedere ad un byte i 16 bit di indirizzo vengono ancora posti su $AD_{15} - AD_0$; poi gli 8 bit da inviare in uscita sono posti su $AD_{15} - AD_8$ *e* su $AD_7 - AD_0$; mentre per riceverli in ingresso sono letti da $AD_{15} - AD_8$ *o* da $AD_7 - AD_0$ in funzione del fatto che l'indirizzo sia pari o dispari.

Unità di Gestione della Memoria (MMU)

Che cosa significa questo per la MMU? Essa non può vedere $AD_7 - AD_0$ e quindi risponde solamente a $AD_{15} - AD_8$. La MMU risponde alle 64 configurazioni 00-3F presenti su $AD_{15} - AD_8$, utilizzandole per indirizzare i propri registri interni, ma fa questo solamente quando $ST_3 - ST_0$ indicano uno stato di «I/O speciale» ed è valido il segnale di selezione dispositivo. Il progettista di sistemi deve fornire una logica esterna per la selezione del dispositivo. Questa logica sarà più o meno complessa a seconda del numero di MMU presenti nella configurazione. La disposizione ovvia è usare il byte di valore più alto dell'indirizzo ($AD_{15} - AD_8$) per indirizzare il registro interno ed il byte di valore più basso ($AD_7 - AD_0$) per comunicare alla logica di selezione-dispositivo quale è il dispositivo selezionato. Il progettista può definire come usare $AD_7 - AD_0$, ma A_0 dovrà essere sempre zero. Se A_0 non è zero, la CPU si aspetta di ricevere in ingresso da $AD_7 - AD_0$ un byte, perché gli indirizzi con A_0 uguale ad uno sono dispari. Dal momento che la MMU può inviare segnali solamente su $AD_{15} - AD_8$, tutti gli indirizzi della MMU devono essere pari; questo significa che il progettista può utilizzare le sette linee $AD_7 - AD_1$ per codificare fino a 128 diverse MMU. Se si usano meno di otto MMU, ognuna di queste può essere identificata da uno dei bit compresi tra AD_7 e AD_1. Questa struttura è vantaggiosa in termini di semplicità ed inoltre le diverse MMU possono inviare contemporaneamente la stessa istruzione. Questo potrebbe essere importante in quei casi in cui due MMU diverse devono avere alcuni valori identici predisposti nei loro registri interni. In quest'ultimo caso se si verifica una interruzione tra la predisposizione di una MMU e la predisposizione dell'altra si potrebbe avere qualche guaio. Nella pubblicazione della Zilog di David Stevenson «*Introduzione all'Unità di Gestione della Memoria Z8000 – Informazioni Pratiche* (An Introduction to the Memory Management Unit – Tutorial Information)» viene descritto in dettaglio un esempio di configurazione che utilizza questo sistema di indirizzamento.

Fino ad ora non abbiamo discusso i registri o il funzionamento di una MMU. Adesso daremo una breve descrizione di essi.

La MMU contiene un insieme di 64 registri e 32 bit per la descrizione del segmento, tre registri di controllo ad un byte e sei byte per lo stato di «violazione dell'informazione».

L'insieme dei registri di descrizione dei segmenti è indirizzato dagli ultimi sei bit del numero di segmento ($SN_5 - SN_0$). Un bit in uno dei registri di controllo determina se i 64 segmenti sono compresi tra 0 e 63 o tra 64 e 127; perciò, $SN_6 = 0$ o $SN_6 = 1$. Ciascun descrittore di segmento contiene un indirizzo di base (che specifica i bit 23-8 della base fisica del segmento), un campo di limitazione che indica il numero di blocchi di 256 byte di memoria fisica realmente assegnati al segmento (non tutti i segmenti utilizzano tutti i 64 Kbyte disponibili) ed otto bit di «attributi» che indicano se il segmento:

- è stato esaminato
- è stato cambiato
- deve ricevere un trattamento speciale ad uso dello stack
- non permette accessi in DMA
- deve essere solo eseguito
- non permette accesso alla CPU
- non permette l'accesso di modo normale
- deve essere solo letto.

I tre registri di controllo ad un byte contengono otto bit che funzionano come un registro indirizzo di un byte per accedere all'insieme del descrittore del segmento, tre bit per specificare quale delle linee $AD_{15} - AD_8$ posizionerà questa MMU nella «ragione» al verificarsi di una trap di segmento e cinque bit che dicano se la MMU:

- farà fluttuare le sue linee di indirizzo fisico
- passerà direttamente gli indirizzi della sua estensione alla memoria fisica senza convertirli
- risponderà ai segmenti 0-63 oppure 64-127
- risponderà al modo sistema, al modo normale o ad entrambi.

Gli otto bit rimanenti non sono usati.

I sei registri contenenti l'informazione di violazione ad un byte contengono le informazioni salvate che si riferiscono alla causa che ha originato la trap:

- I 15 bit superiori dell'indirizzo di segmento a 23 bit che ha causato la trap.
- I 15 bit superiori dell'indirizzo di segmento a 23 bit dell'ultima istruzione corretta.
- Le linee di stato $N/\overline{S}$, $R/\overline{W}$ e $ST_3 - ST_0$ all'istante in cui si è verificata la trap.
- Otto bit che descrivono il tipo di violazione:

- quattro corrispondenti alle proibizioni predisposte nei bit di attributo (tutti escluso il DMA)
- uno per il superamento della larghezza specificata dal limitatore di campo
- due per i vari tipi di supero dello stack.
- un bit «fatale», usato quando si accumulano gli errori.

Non forniremo la struttura dettagliata dei registri e degli indirizzi interni della MMU. Per quanto riguarda queste informazioni consigliamo il lettore di riferirsi alla documentazione del costruttore.

La Famiglia di Componenti dello Z8000

La famiglia di componenti dello Z8000 è costituita dai seguenti dispositivi:

- CPU Z8000
- Unità di Gestione della Memoria
- Controllore delle Comunicazioni Seriali
- Dispositivo Contatore/Temporizzatore e I/O Parallelo
- Buffer First In, First Out, CPU/CPU o CPU/Periferica
- Controllore DMA
- Controllore CRT (Video)
- Controllore Floppy Disk
- Controllore Universale di Periferiche basato su Z8.

I/O Seriale

Nell'ultimo capitolo abbiamo parlato della trasmissione sincrona seriale dei bit dei dati. Abbiamo esaminato alcuni dei problemi coinvolti nella *conversione seriale-parallelo,* esempio il prelievo di caratteri di otto bit dalla stringa di caratteri. Questa funzione, ed il suo complemento, *la conversione parallela-seriale* vengono generalmente realizzate tramite un circuito integrato. Un circuito di questo tipo, utilizzabile con lo Z8000, è lo Z80A-SIO. Esso rappresenta un esempio di come sono fatti questi circuiti. Questo componente è adeguatamente descritto nella documentazione del costruttore, per cui nel seguito noi non lo descriveremo in dettaglio.

Lo scopo di un circuito seriale di I/O è di permettere la trasmissione parallela da parte della CPU e la trasmissione seriale da parte del dispositivo periferico. Siccome esistono degli standard generalmente accettati (esempio RS-232C), per i dispositivi periferici seriali di I/O, si richiede una trasmissione seriale da parte del dispositivo periferico. Questo tipo di trasmissione è anche più adatto a trasmissione su lunghe distanze (richiede un minor numero di cavi ed elimina i problemi dovuti alla sincronizzazione di segnali trasmessi in parallelo). Per la trasmissione seriale si possono

usare i modem e le linee telefoniche. La trasmissione seriale è standardizzata e dotata di molte opzioni: baud-rate, numero di bit per carattere, numero di bit di stop, parità dei caratteri, conversione per i blocchi ed i formati dei campi dati trasmessi, controllo dell'errore longitudinale e molti altri. I primi circuiti di I/O seriale (esempio UART) fornivano alcune di queste opzioni mediante segnali presenti ai loro piedini di ingresso. Un moderno circuito, tipo lo Z80A-SIO, possiede dei registri interni che possono essere programmati tramite comandi inviati dalla CPU.

I/O Parallelo

Mentre per l'I/O seriale esistono delle convenzioni standard, per l'I/O parallelo c'è molto poco. L'interfaccia parallela di I/O permette ai dispositivi di essere interfacciati in modo standard, in particolare rispetto alla convenzione usata dalla CPU per le interruzioni e i riconoscimenti. Lo Z80A-PIO è un circuito di questo tipo che può essere usato con lo Z8000.

Circuiti Contatori/Temporizzatori

Lo scopo principale di un circuito contatore/temporizzatore è di permettere delle funzioni di temporizzazione, come quelle necessarie per la trasmissione sincrona seriale di bit, per poter gestire ad esempio una trasmissione sincrona seriale senza il completo controllo della CPU. La CPU programma il circuito in modo da ricevere una interruzione ad intervalli determinati. Quando si verifica una interruzione, la CPU fa tutto quello che deve fare per gestire l'interruzione (esempio campiona l'ingresso nel centro della cella del bit). Lo Z80A-CTC è un circuito di questo tipo e può essere usato con lo Z8000.

Non entreremo in ulteriori dettagli nell'analisi della famiglia di componenti dello Z8000; è sufficientemente ovvia l'utilizzazione di un Controllore per Floppy Disk o CRT (video). La ragione di un Buffer First In First Out è quella di permettere a dei dispositivi di diversa velocità di comunicare tra di loro senza rallentare il più veloce. Il Controllore per l'Accesso Diretto alla Memoria (DMA) consente di trasferire dei dati alla velocità della memoria senza l'intervento della CPU. Quest'ultimo è utilizzato per dispositivi che hanno una velocità di trasferimento molto elevata (esempio dischi).

Alcune caratteristiche non pianificate

Ci sono varie caratteristiche dello Z8000 che trovano una utilizzazione

al di fuori di quelle applicazioni per cui sono state originariamente sviluppate. Per esempio, nel Capitolo IV abbiamo parlato di come sono utilizzate le trap causate da «istruzioni non implementate». Un'altra caratteristica promettente è il registro di REFRESH (vedasi Figura 2.10).

Supponete, per esempio, che il campo RATE sia posto a 50 e la frequenza di clock a 4MHz; quindi il campo ROW viene incrementato di 2 ogni 50 μsec. Perciò, se volete eseguire un'attesa di 10 msec, tutto quello che dovete fare è controllare il valore corrente del campo ROW, chiamate una qualsiasi altra routine che deve essere eseguita, poi ritornate indietro ad aspettare fino a quando il campo ROW è stato incrementato di un valore totale di 400 sopra il valore originale. Non dovete più preoccuparvi della quantità di tempo realmente richiesta dalla routine chiamata (supponendo che sia minore di 10 msec). Sono possibili molte variazioni su questo tipo di applicazione.

Utilizzando il registro REFRESH, ogni volta che si verifica un ciclo di rinfresco, si usano tre cicli di clock (nell'esempio sopra ogni 50 μs). Questo ciclo di rinfresco può essere realmente usato per realizzare un clock: una logica esterna può generare un'interruzione ogni volta che il contatore ROW raggiunge 400 in un ciclo di rinfresco e una routine di interruzione può aggiornare un clock software di 10 ms e sottrarre 400 dal valore di ROW. Diversamente dall'esempio precedente, in questo caso i circuiti di rinfresco debbono essere dedicati a questa applicazione.

ESERCIZIO 2: (a) Perchè si sottrae 400 dal valore di ROW? Perchè semplicemente non lo si azzera?

(b) Scrivete la codifica per sottrarre 400 dal valore di ROW. Che cosa succede se il contatore ROW ha già raggiunto 512 ed ha riciclato a zero prima che venga gestita l'interruzione? Come eliminate la possibilità di disturbo del campo RATE e del bit di abilitazione? Che cosa fate se ha già raggiunto 288 (800 mod 512)?

Anche la prestazione della sincronizzazione multi-micro può essere utilizzata in modo non standard. Può essere usata come una porta di I/O ad un bit. L'istruzione MREQ può essere utilizzata per generare impulsi di durata prefissata.

ESERCIZIO 3: (a) Scrivete la codifica per generare un impulso di 100 ms su MO.

(b) Scrivete la codifica per far entrare il valore MI nel bit meno significativo di un registro. (Suggerimento: ricordate TCC.)

Questo esercizio conclude la nostra discussione sui componenti periferici dello Z8000. Il lettore che vuole proseguire ulteriormente lo studio di questo argomento è indirizzato al volume Sybex di prossima pubblicazione: *Z8000 Applications Book*. (Applicazioni dello Z8000).

Capitolo VIII

Esempi di Programmi di Utilità

Pensiamo a tutti gli esempi di sistemi a calcolatore che non servono molto l'utente: un grande sistema di elaborazione in batch che richieda che i parametri di controllo dei lavori, come il limite di tempo o il massimo numero di linee di uscita, siano specificati in ottale; un piccolo sistema interattivo che non vi permette di digitare in anticipo, quando voi conoscete già la risposta, prima che siano formulate le domande; il prolificare di grandi e piccoli sistemi che non possono trattare le date in una forma comprensibile all'utente. La cosa più triste da dire è che la maggior parte di questi sistemi sono stati progettati da programmatori che si sono sicuramente preoccupati di renderli funzionali.

I problemi principali sono il tempo e gli strumenti a disposizione. Si è impiegato molto ad inventare la ruota, come vi diranno quelli di noi che lo hanno fatto molte volte. O come disse una persona «Andremo più in alto mettendoci uno sulle spalle dell'altro invece di cercare di pestarci i piedi». Naturalmente, un libro non può risolvere molti problemi, ma se userete alcune delle ruote di questo capitolo, invece di reinventarle, potrete avere un po' di tempo libero per cercare di rendere un poco più facile la vita dell'utente.

Tutti i programmi di questo capitolo sono scritti per lo Z8000 non-segmentato. Per il funzionamento segmentato occorrono versioni analoghe ma con diverse locazioni di memoria o registri per gli indirizzi.

Inizializzazione degli I/O

Nel Capitolo VI abbiamo presentato la routine di gestione delle interruzioni TYOUT e TYIN. Ora dovremo scrivere la routine TYIO che inizia-

lizza le operazioni sviluppate da TYOUT, ma prima dobbiamo farci una idea migliore di come abbiamo intenzione di usare questo meccanismo.

TYIO è un programma di utilità (utility) per i programmi applicativi che usano il terminale. Viene chiamata per fare una delle due seguenti cose:

(1) Ingresso di una linea di caratteri dalla tastiera.

(2) Invia in uscita alla telescrivente o al video una linea di caratteri.

Per esempio, il programmatore può desiderare inviare in uscita il messaggio

«Enter your password»
(Digitate la vostra parola d'ordine)

e poi accettare una risposta, digitata, chiusa con un carriage return (carattere CR). La prima di queste operazioni corrisponde al punto (2) sopra, la seconda al punto (1).

Far entrare una linea di caratteri dalla tastiera comporta varie difficoltà, la prima delle quali è l'eco. Se l'operatore ha digitato una «A», allora il programma deve inviare una «A» indietro alla stampante/video. L'eco deve aspettare che finisca ogni operazione di uscita precedentemente iniziata. D'altra parte, l'operatore può aver digitato la «A» prima che il programma formuli la domanda; per cui la routine di gestione dall'interruzione da tastiera deve prendere i caratteri e memorizzarli fino a quando il programma richiede un ingresso.

La gestione dei caratteri speciali è un'altra delle difficoltà legate all'ingresso. Per esempio, il carattere RUBOUT ha bisogno di un eco speciale; ha bisogno di far si che il carattere precedente venga rimosso dal buffer in cui viene formata la linea d'ingresso. L'eco dipende dal fatto che il visualizzatore del terminale sia un video o una stampante (la routine TYOUT del capitolo VI suppone sia un video). Se non c'è alcun carattere da cancellare viene soppresso.

Nell'esempio sopra riportato, immaginate che l'operatore inizi a digitare la parola d'ordine, prima che sia richiesta dal programma, si fermi nel mezzo di questa operazione per ricordarsi qualcosa, poi infine finisca la digitazione dopo che il calcolatore ha riguadagnato il precedente ritardo. La Figura 8.1 illustra una possibile sequenza di eventi.

In uno degli esercizi del Capitolo VI abbiamo accennato ad alcuni dei problemi che potrebbero nascere se le routine di interruzione TYOUT e TYIN si interrompessero l'una con l'altra. Fortunatamente possiamo essere sicuri che questo non si verifica. TYOUT e TYIN devono interrompere TYIO, TYIO dovrà aspettare che FLAG venga posizionato a DO-

EVENTO	BUFFER DI INGRESSO	LINEA DI INGRESSO	VIDEO
L'utente digita: G, U, M	G,U,M		_
Il programma chiede «Introdurre la Password» L'utente digita: R, RUBOUT.	G,U,M,R, RUBOUT		INTRODURRE LA PASSWORD
L'interruzione finale si verifica dopo che il messaggio è stato inviato in uscita. FLAG è posizionato a DONE. Il programma richiede un ingresso. Inizia l'elaborazione. L'utente digita: D, R, O, P.	RUBOUT,D,R, O,P	G,U,M,R	INTRODURRE LA PASSWORD GUMR _
TYOUT preleva il carattere RUBOUT dal buffer di ingresso ed opera su di esso, toglie R dalla linea di ingresso ed invia in eco BS, SPACE, BS.	D,R,O.P	G,U,M	INTRODURRE LA PASSWORD GUM _
TYOUT elabora i successivi quattro caratteri. L'utente non ha ancora premuto il RETURN, per cui l'ingresso non è finito ed il buffer è vuoto. FLAG è posizionato a SLEEP		G,U,M,D,R, O,P	INTRODURRE LA PASSWORD GUMDROP _
L'utente preme il RETURN. Quando TYIN pone il RETURN nel buffer di ingresso, posiziona nuovamente FLAG ed INPUT e salta TYOUT.	RETURN	G,U,M,D,R, O,P	INTRODURRE LA PASSWORD GUMDROP _
TYOUT preleva il RETURN dal buffer di ingresso. Invia come eco i caratteri CR, LF e memorizza un NUL nel buffer di linea. FLAG è posizionato a DONE.		G,U,M,D,R, O,P,NUL	INTRODURRE LA PASSWORD GUMDROP

Figura 8.1 — Sequenza degli Eventi di un Ingresso da Tastiera

NE. Questo verrà fatto guardando e riguardando FLAG fino a quando non sarà posizionato a DONE; se non fosse possibile inviare interruzioni, questo sarebbe un loop infinito siccome non sarebbe possibile cambiare la configurazione di FLAG.

Ricordatevi che la causa dell'interazione era dovuta alle due diverse copie del contenuto prelevate dall'area di memoria situata in CONBLK e caricata nei registri. Ciascun programma che preleva una copia del contesto riscrive la sua copia quando è stato eseguito, quindi una delle copie deve essere completata mentre viene scritta sull'altra — ogni cambiamento fatto sulla prima copia sarebbe perso quando la seconda copia viene scritta sopra di questa.

Se TYIO usa TYENT e TYEXIT per predisporre e salvare il contesto e se TYIO viene interrotta da TYOUT o TYIN, allora verrà perso ogni cambiamento fatto da TYOUT o TYIN sul contesto (esempio pone FLAG = DONE). Siccome TYIO deve aspettare che FLAG assuma il valore DONE, questa attesa può essere realizzata senza leggere ed aggiornare il blocco del contesto. Dopo che FLAG è stata posta uguale a DONE, TYIO è libera di usare il blocco del contesto perchè non ci dovrebbero essere interruzioni di TYOUT e le interruzioni di TYIN lasciano inalterato il contesto.

La Figura 8.2 mostra la routine di TYIO. La routine TYWAIT, in fondo, guarda la copia di FLAG nella memoria, che è il primo byte del contesto localizzato con CONBLK. Questo tiene conto del contenuto ***MUST BE FIRST*** (Deve essere per primo) nelle definizioni di TYENT e TYEXIT.

ESERCIZIO 1: Considerate questo possibile approccio per evitare di avere due routine con la copia del contesto. Supponete che la prima parola di CONBLK sia un *semaforo*: quando ha valore −1, non si può usare il contesto; quando è zero, si può prendere ed usare il contesto. La sequenza

```
X:  TSET CONBLK; JR MI,X
```

deve precedere ogni lettura del contesto; ma

```
CLR CONBLK
```

deve seguire la riscrittura del contesto.

Quindi, TYENT può essere scritta secondo le linee

```
TYWSM:  TSET CONBLK; JR MI,TYWSM
```

che precede il prelievo del contesto; TYEXIT può avere

```
CLR CONBLK
```

```
!Inizializzatore del I/O per il terminale (versione non-segmentata)

  CALL TYIO; R1 = indirizzo della tabella degli argomenti
  Ritorna con C = 0 se è inizializzata
              C = 1 se c'è un comando non valido
  I/O viene eseguito sotto interruzione.

  CALL TWAIT
  Ritorna dalla routine quando è finita la precedente operazione.

  Formato della tabella degli argomenti:

   Indirizzo byte      Per richiesta di ingresso   Per richiesta di uscita
!
   PFLAG = 0     !#INPUT                          #OUTPUT                    !
   PADR  = 2     !Indirizzo della linea           Indirizzo della stringa di uscita!
   POUTA = 4     !Indirizzo dell'I/O di uscita    Uscita dell'indirizzo di I/O  !
   PRING = 6     !Indirizzo di RING               (non usato)                   !
   !Tabella delle
   lunghezze (byte): !  LINTAB = 8;              LOUTAB = 6                    !

TYIO:   CALR TYWAIT                      !Attendi «done»!
        CALR TYENT                       !Posiziona il contesto!
        LDB FLAG,R1(#PFLAG + 1)          !Posiziona il FLAG dal-
                                         la tabella!
        LD IOUT,R1(#POUTA)               !  e l'uscita dell'indirizzo
        CPB FLAG,#INPUT; JR NE,TY1       !Se l'ingresso:  !
          LD RING,R1(#PRING)             !  Poni RING=1!
          LD LINE,R1(#PADR)              !     & LINE=1!
        JR TYOKX
TY1:    CPB FLAG,#OUTPUT; JR NE,TYERX    !Altrimenti se l'uscita: !
          LD PTR,R1(#PADR)               !  Poni ad 1 PTR!
                                         !Se c'è un errore!
TYOKX:  RESFLG C; CALR TYEXIT            !OK per l'uscita: C=0!
        SC #IOGO; RET                    !Inizializza I/O!
TYERX:  SETFLG C                         !Errore: C=1!
        LDB FLAG,#DONE                   !FLAG = DONE!
        CALR TYEXIT; RET
!Routine d'attesa!
TYWAIT: CPB CONBLK,#DONE; JR NE,TYWAIT; RET
```

Figura 8.2 — Inizializzatore dell'I/O per il Terminale

inserita giusto dopo aver ristabilito il contesto. Tutte le routine che chiamano TYENT e TYEXIT accedono al blocco del contesto in modo automatico.

Adesso TYWAIT può essere scritto:

```
TYWAIT:  CALL TYENT
         CPB FLAG,#DONE; JR EQ,TYWEX
         CALL TYEXIT; JR TYWAIT
TYWEX:   CALL TYEXIT; RET
```

Se si verifica una interruzione a TYIN o TYOUT tra le chiamate TYWAIT, TYENT e TYEXIT, la routine di interruzione chiamerà TYENT e rimarrà nel loop di TYWSM fino a quando TYWAIT termina il suo test e restituisce il contesto. Poi continua l'elaborazione dell'interruzione, e TYWAIT non altererà i cambiamenti fatti da TYIN e TYOUT.

Perchè questo non funziona? La situazione che lo fa sbagliare è chiamata «*nodo mortale*».

Ci sono altri punti di Figura 8.2 da evidenziare. Gli argomenti chiamati vengono passati tramite una tabella indirizzata da R1. Le posizioni degli argomenti nella tabella sono definite e contemporaneamente commentate e utilizzate dentro la routine con il modo di indirizzamento tramite base. Siccome questa routine è scritta per il funzionamento non-segmentato, avremmo potuto usare l'indirizzamento indicizzato (esempio LD IOUT,POUTA(R1) invece di LD IOUT,R1(#POUTA)), ma questo avrebbe complicato il nostro lavoro quando avremmo voluto scrivere la analoga routine per il funzionamento in modo segmentato. In effetti, se avessimo dovuto fornire la definizione

TABADR = R1

all'inizio della routine (forse sulla stessa linea come «R1 = adr della tabella argomenti»), e scrivere, per esempio,

LDB FLAG,TABADR(#PFLAG + 1)

invece di

LDB FLAG,R1(#PFLAG + 1)

in questo caso il solo cambiamento necessario a *questa* routine, per poterla utilizzare in modo segmentato, è una ridefinizione del tipo

TABADR = RR2

Ma naturalmente sarebbe necessario riscrivere TYENT e TYEXIT per fornire una diversa allocazione del registro.

> ESERCIZIO 2: Scrivete le versioni di TYENT e TYEXIT che forniscono la giusta allocazione del registro nel funzionamento segmentato. Un sistema simile a quello definito nella TABADR menzionata sopra può essere usato per localizzare i necessari cambiamenti del codice sorgente?

Per ultimo, notate che l'istruzione

SC #IOGO

segue la chiamata di TYEXIT. Ricordatevi che questo è un mezzo (da discutere nel Capitolo IX) per simulare una interruzione di una uscita per un terminale, siccome non sono stati ancora inviati dei caratteri al terminale in risposta all'interruzione finale dell'ultimo programma di uscita (quello che fa sì che FLAG sia posto uguale a DONE). Perchè esso segue la chiamata di TYEXIT?

Routine di Gestione del Buffer ad Anello

Nel Capitolo VI abbiamo usato un buffer ad anello come buffer d'ingresso da terminale. Un buffer ad anello è semplicemente un buffer first in first out e può essere usato in molte applicazioni. Le routine di base per la gestione sono GETRNG e PUTRNG. Per il metodo qui descritto viene richiesta, come conseguenza diretta, una inizializzazione che non descriveremo in dettaglio.

La Figura 8.3 mostra le sequenze di chiamata di queste routine e il formato dei buffer ad anello che esse gestiscono. La parte di memorizzazione e un insieme di n byte. Viene chiamato «anello» perchè il byte 1 viene considerato come il byte successivo al byte n. Ci sono due puntatori che si muovono lungo l'anello. Il puntatore di put (mettere) punta alla successiva posizione libera, dove verrà aggiunto il prossimo elemento; il puntatore di get (prendere) punta all'elemento più vecchio del buffer, il prossimo da togliere.

Una posizione di un byte viene considerata vuota se contiene uno zero (per cui non si possono memorizzare degli elementi di valore zero; questo è importante per i caratteri). Se il puntatore-get sta puntando ad un elemento zero, il buffer è vuoto. Se il puntatore-put *non* sta puntando ad un elemento zero il buffer è pieno. Ogniqualvolta GETRNG toglie un elemento per mezzo del puntatore-get, lo sostituisce con uno zero. Questa procedura è illustrata in Figura 8.5.

```
!Routine per il Buffer ad anello (versione non-segmentata)

  CALL GETRNG; R1 = indirizzo dell'anello
  La routine ritorna con C = 0, e il carattere successivo in RL0 o C = 1 se il buf-
  fer è vuoto.

  CALL PUTRNG; R1 = indirizzo dell'anello; RL0 = carattere.
  La routine ritorna con C = 0 se il carattere è memorizzato, C = 1, trascurando
  il carattere, se il buffer è pieno.

  Formato di un anello:
        Prima parola = n (numero di byte memorizzati nell'anello)
        !RGET = 2     !2° parola = puntatore – Get (da 0 a n–1)!
        RPUT = 4      !3° parola = puntatore – Put (da 0 a n–1)!
        RDAT = 6      !Inizio della memorizzazione di n byte!

!Registri utilizzati:                    R0,R1 comunicano con il chiamante!
!   RSCR     = 2                         !R2,R3 sono di tipo scratch!
    RNGCON   = R4                        !Blocco dei registri di contesto!
      RSIZ   = RNGCON                    !Dimensioni dell'anello di memorizzazione!
      GETP   = RNGCON+1                  !Puntatore di Get!
      PUTP   = RNGCON+2                  !Puntatore di Put!
    RNGRT    = RNGCON+3                  !Ultimo chiamante-non è nella testata del-
                                         l'anello!
    NRNCX    = 3                         !Numero di registri per la testata!
    RNCXB    = 2*NRNCX                   !Numero di byte da memorizzare nello
                                         stack!
    NRNRG    = NRNCX+RSCR                !Registri per la testata e scratch!
    RNRGB    = 2*NRNRG                   !Numero totale di byte!

!Routine per il salvataggio e il ripristino del contesto:
  CALL RNGENT (salva i registri e posiziona il contesto) – C è conservata
  CALL RNGEX  (ripristina i registri) – C è conservata
```

Figura 8.3 — Definizioni e Formati dei Buffer ad Anello

Noi usiamo qui un approccio leggermente diverso a causa dei problemi avuti col blocco di contesto delle routine di I/O del terminale. RNGENT legge il blocco di contesto nei registri partendo dall'inizio dell'anello. RNGEX *non riscrive* fuori l'intero contesto. Invece GETRNG scrive il

puntatore-get aggiornato nella memoria, mentre PUTRNG scrive l'aggiornamento del puntatore-put.

L'utilizzo di un indice (da 0 ad n − 1) per i puntatori rende l'anello, come struttura dati, indipendente dalla sua locazione in memoria ed anche indipendente dal tipo di segmentazione. Naturalmente, i programmi che gestiscono gli anelli suppongono un funzionamento non-segmentato (esempio utilizzando R1 come un registro indice) anche se gli anelli in se stessi non contengono indirizzi.

ESERCIZIO 3: (a) Riscrivete la routine di I/O per terminale in modo tale che TYEXIT non prelevi il blocco di contesto. Fornite le definizioni del tipo RGET, RPUT, RDAT (Figura 8.3) per FLAG, PTR e di altri elementi che cambiano, e usate

```
LDA R2,CONBLK
```

per predisporre un indirizzo base. Utilizzate il modo di indirizzamento su base per permettere a TYOUT e TYIO di aggiornare gli elementi cambiati. Il registro di contesto dovrà essere spostato in giù di una posizione, esempio FLAG = R3, CHAR = RL3, PTR = R4, ecc.

(b) Siccome TYIN non modifica il contesto, questa potrebbe ridurre il tempo utilizzato per elaborare le interruzioni da tastiera. Calcolate i tempi tipici richiesti da TYIN per la versione corrente e la vostra della routine.

(c) Adesso supponete che si verifichi una interruzione di uscita per terminale mentre TYIN sta eseguendo al suo interno una chiamata a PUTRNG. Che effetto può avere questo?

La routine di Gestione della Linea di Memorizzazione Caratteri

Le routine ADLINE e BACKUP che gestiscono la linea d'ingresso composta da TYOUT, quando elaborano i caratteri provenienti dal buffer ad anello, sono estremamente semplici.

ESERCIZIO 4: State progettando un programma e incominciate a scrivere la codifica di Figura 8.6 esattamente da dove era stato lasciato quando il programmatore due mesi prima era passato ad un altro lavoro.

(a) Definite le convenzioni di chiamata per BACKUP e ADLINE. (Suggerimento: trovate dove sono chiamate nel Capitolo VI).

(b) Qual è il formato di una linea?

(c) Che cosa succede se ADLINE viene chiamata quando la linea è piena?

La Routine di Conversione

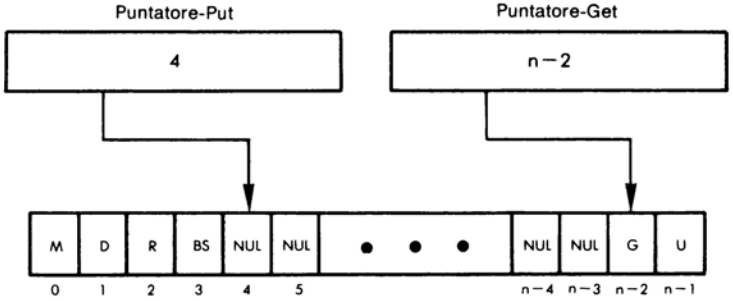
TRAN è il programma più semplice e più utile di questo capitolo. Esistono molte possibili varianti, ma nella sua forma più semplice TRAN a-

```
!Routine per l'anello: RNGENT, RNGEX, GETRNG, PUTRNG!

RNGENT:  EX RNGRT,@SR                    !Scambia il ritorno in
                                         RNGRT!
         DEC SR,#RNRGB                   !Salva altri registri - escluso
                                         R0, R1!
         LDM @SR,R2,#NRNRG
         LDM RNGCON,@R1,#NRNCX           !Carica il contesto!
         JP @RNGRT                       !USCITA!

RNGEX:   POP RNGRT,@SR                   !Preleva il ritorno per
                                         RNGRT!
         LDM R2,@SR,#NRNRG               !Ripristina i registri salvati!
         INC SR,#RNRGB
         EX RNGRT,@SR; RET               !Ripristina RNGRT e l'uscita!

GETRNG:  CALR RNGENT                     !Struttura il contesto!
         LD R2,GETP; INC R2,#RDAT        !Sistema l'informazione per
                                         la testata dell'anello!
         LDB RL0,R1(R2); TESTB RL0       !Carica il carattere; controlla se è vuoto il buffer!
           JR Z,RGER
         CLRB RL3; LDB R1(R2),RL3        !Sostituisce il carattere con
                                         zero!
         INC GETP                        !Incrementa il puntatore-
                                         Get!
         CP GETP,RSIZ                    !Riduce il modulo n del puntatore-Get!
           JR LT,GPUP
         SUB GETP,RSIZ
GRUP:    LD R1(#RGET),GETP               !Aggiorna il puntatore-Get
                                         in memoria!
RGOK:    RESFLG C; CALR RNGEX; RET       !OK per uscite: C = 0!
RGER:    SETFLG C; CALR RNGEX; RET       !Vuoto/Pieno: C = 1!

PUTRNG:  CALR RNGENT                     !Struttura il contesto!
         LD R2,PUTP; INC R2,#RDAT        !Sistema l'informazione per
                                         la testata dell'anello!
         LDB RL3,R1(R2); TESTB RL3       !Guarda se la posizione è libera!
```

Figura 8.4 — Routine di Gestione del Buffer ad Anello (segue)

```
        JR NZ,RGER
        LDB R1(R2),RL0; INC PUTP      !Memorizza il carattere, incrementa il puntatore!
        CP PUTP,RSIZ                  !Riduce il mod n del puntatore!
        JR LT,PPUP
        SUB PUTP,RSIZ
PPUP:   LD R1(#RPUT),PUTP             !Aggiorna il puntatore-put in memoria!
        JR RGOK                       !Prende l'OK per l'uscita!
```

Figura 8.4 — Routine di Gestione del Buffer ad Anello



L'anello contiene i caratteri:

G,U,M,D,R,BS

La successiva chiamata di GETRNG fornirà G, piazzerà un NUL nella posizione n − 2 e aggiornerà il puntatore-Get per contenere n −1.

La successiva chiamata di PUTRNG metterà un carattere nella posizione 4 e aggiornerà il puntatore-Put al valore 5.

Figura 8.5 — Funzionamento di un Buffer ad Anello

nalizza una tabella di coppie di elementi cercando un elemento uguale al primo elemento di ciascuna coppia; se l'uguaglianza viene verificata restituisce il secondo elemento della coppia. Questa tecnica è conosciuta come *ricerca associativa.*

A prima vista questa routine non sembra molto piacevole da usare. Nel Capitolo VI abbiamo visto come questa routine potrebbe essere usata,

```
!Esempio di come non scrivere un programma!

BACKUP:  PUSHL @SR,RR2   !Salva R2,R3!
         LD R2,R1(#2)    !Preleva il puntatore!
         TEST R2         !Lo controlla!
         JR MI,LNER      !Vuoto!
         LD R3,#4        !Spiazzamento della parte di memorizzazione
                         dal dato della linea!
         ADD R3,R2       !Lo somma al puntatore!
         LDB RL0,R1(R3)  !Preleva il carattere!
         DEC R2          !Decrementa il puntatore!
         LD R1(#2),R2    !Lo mette da parte!
LNOK:    RESFLG C        !Azzera C!
LNEX:    POPL RR2,@SR    !Ripristina R2,R3!
         RET             !Ritorno!

ADLINE:  PUSHL @SR,RR2   !Salva R2,R3!
         LD R2,R1(#2)    !Preleva il puntatore!
         LD R3,R2        !Lo pone in R3!
         INC R3          !Incrementa R3!
         CP R3,@R1       !Lo confronta con la dimensione della linea!
         JR EQ,ADL1      !Linea piena!
         LD R2,R3        !Usa R3!
         CLR R3          !Zero significa che non era pieno!
ADL1:    LD R1(#2),R2    !memorizza il puntatore!
         ADD R2,#4       !Somma lo spiazzamento del dato!
         LDB R1(R2),RL0  !Memorizza il carattere!
         TEST R3         !Controlla se la linea è piena!
         JR Z,LNOK       !Non è piena!
LNER:    SETFLG C        !Pone C = 1!
         JR LNEX         !Va all'uscita!
```

Figura 8.6 — BACKUP e ADLINE

basandosi sul valore di una flag o di un carattere, per realizzare un trasferimento ad una possibile locazione. Più avanti ci saranno altri esempi. Una volta che sarete abituati ad usare TRAN, troverete ad ogni momento delle nuove applicazioni in cui utilizzarla.

La Figura 8.7 illustra la routine TRAN. Ci sono molti modi per variare questa routine di base: dimensionare diversamente gli elementi, definire dei valori diversi delle dimensioni dell'elemento convertito, prevedere azioni diverse sui guasti.

Secondo l'autore l'idea di canonizzare la tecnica della ricerca associativa in una routine di tipo generale come TRAN è dovuta a R.S. Langer.

> ESERCIZIO 5: Esiste un suffisso tipico di ciascuno dei 31 numeri dei giorni di un mese di un calendario (esempio 1st, 2nd, 3rd, 4th); costruite una tabella per TRAN che dia questi suffissi. Per farla breve, fate in modo che i valori di default siano i più comuni.

TRAN è il nostro ultimo legame con il Capitolo VI. Il prossimo esempio mostrerà come viene utilizzata la routine TYIO.

Programma di Interazione con il Terminale

Se un essere umano deve interagire con un calcolatore, allora il sistema

```
!Routine di conversione (versione non-segmentata)

  CALL TRAN; R0 = elemento da convertire
             R1 = indirizzo della tabella
    La routine ritorna con C = 0 e R0 = elemento convertito, se questo viene
                           trovato
                           C = 1 e R0 = valore di default, se non viene trova-
                           to l'elemento
                           R1 = indirizzo dell'elemento fornito dalla routine
                           all'uscita (elemento convertito o valore di default)
  Formato della tabella:
    Elemento 1; Valore di conversione dell'elemento 1
                •
                •
                •
    Elemento n; Valore di conversione dell'elemento n
             0; Default

!
TRAN:     TEST @R1; JR EQ,TRANF  !Cerca il terminatore!
          CP R0,@R1; JR EQ,TRANS !Poi controlla l'uguaglianza!
          INC R1,#4; JR TRAN     !Preleva la coppia successiva!
TRANF:    SETFLG C; JR TROUT     !Non trovato!
TRANS:    RESFLG C               !Trovato!
TROUT:    INC R1,#2; LD R0,@R1   !R1 punta al valore!
          RET
```

Figura 8.7 — La Routine TRAN per la Ricerca Associativa

dovrebbe fornire al programmatore gli strumenti necessari per strutturare in modo valido tale interazione. Il programma qui discusso fornisce alcuni elementi di base. Vengono poi indicate, come esercizi, alcune possibili espansioni o miglioramenti. Il programma ridotto all'osso consiste di quattro routine: SAY, ASK, GETCH, BACKCH.

SAY prende l'indirizzo di una stringa terminata e la invia in uscita al terminale.

ASK prende l'indirizzo di una (possibilmente di lunghezza zero) stringa terminata (la domanda), che invia in uscita al terminale, poi accetta l'ingresso (la risposta), che viene memorizzata in un buffer chiamato INLINE. Per ultimo, inizializza il puntatore INLINE per poter essere usato da GETCH e BACKCH.

GETCH prende il carattere successivo da INLINE e lo dà al chiamante; C viene usato per comunicare al chiamante un «end-of-line» (fine linea).

BACKCH «restituisce» il carattere trasmesso dal chiamante. Per esempio, una routine di ingresso di un numero prende caratteri fino a quando vede dei caratteri alfabetici. Poi utilizza BACKCH per restituire il carattere alfabetico.

La Figura 8.8 definisce la struttura della linea caratteri: è analoga ad un anello, ma solamente con un puntatore, che non *si sposta* dalla posizione n − 1 alla posizione 0. Questo puntatore ha anche il valore addizionale −1 corrispondente ad una linea usata. Come nell'anello, una linea è una struttura dati indipendente dal tipo di segmentazione; le routine di gestione sono state scritte supponendo di operare in funzionamento non-segmentato.

La Figura 8.9 mostra le routine GETCH e BACKCH e le due routine ADLINE e BACKUP viste un po' più indietro (vedasi Figura 8.6). Il tipo di rappresentazione di Figura 8.9 detto *top down* (dall'alto verso il basso); per prima cosa ci sono le routine principali, esse considerano la struttura dati ad un elevato livello di generalità. Perciò, le routine principali non fanno assunzioni circa l'implementazione della struttura dati. Esse incrementano o decrementano il puntatore e prelevano o memorizzano un carattere usando le *primitive* LINC, LDEC, LFETCH e LSTORE. Il livello successivo consiste nel realizzare queste primitive in funzione della particolare rappresentazione scelta per la struttura dati. Le routine di scambio del contesto si trovano alla fine. Esse sono virtualmente identiche a quelle delle routine ad anello.

ESERCIZIO 6: Confrontate le routine di Figura 8.6 con quelle di Figura 8.9. Qual è la rappresentazione più chiara? Quale ha i migliori commenti?

```
!Routine di LINE

  CALL ADLINE; RL0 = carattere
                   R1 = indirizzo della linea
  La routine ritorna con il carattere aggiunto (il carattere è scritto sopra l'ultimo
  carattere se la linea è piena)

  CALL BACKUP; R1 = indirizzo della linea
  La routine ritorna con C = 0 e l'ultimo carattere in RL0; o C = 1 se la linea è
  vuota.
  CALL GETCH; R1 = indirizzo della linea
  La routine ritorna con C = 0, RL0 = carattere successivo; o C = 1 e RL0 = 0
se è alla fine.

  CALL BACKCH; RL0 = carattere da sostituire, R 15 è l'indirizzo della linea
  C = 1 se non c'è stata GETCH.

  Formato di una linea:

      Prima parola: n (numero di byte memorizzati in una linea)
!     LPTR = 2          !2ª parola: puntatore (da 0 a n −1, −1 se è vuoto)!
      LDAT = 4          !Inizio della memorizzazione di n byte!

!   Registri usati:
!     LSCR = 1          !R0,R1 per la chiamata R2; è un registro scratch!
      LCON = R3         !Il blocco del contesto viene posto nei registri!
        LNSZ = LCON     !Dimensione della linea
        LNPT = LCON + 1 !Puntatore!
      LNRT = LCON + 2   !Ultimo chiamante-non nella tabella!

      NLNCX = 2                 !Numero dei registri della testata!
      NLNRG = NLNCX + LSCR      !Numero che comprende i registri scratch!
      LNCXB = 2*NLNCX;          !Byte necessari per quanto sopra!
      LNRGB = 2*NLNRG

!Routine per il salvataggio ed il ripristino del contesto:
      CALL LNENT (Salva i registri e posiziona il contesto) — C è conservato
!     CALL LNEX (Ripristina i registri) — C è conservato
```

Figura 8.8 — Definizioni e Formati delle Routine di LINE

```
!Routine di linea: GETCH, BACKUP, ADLINE, BACKCH!

!Line routines: GETCH, BACKUP, ADLINE, BACKCH!

ADLINE:   CALR LNENT
          CALR LINC; CALR LSTORE
          CALR LNEX; RET

BACKUP:   CALR LNENT
          CALR LFETCH; CALR LDEC
          CALR LNEX; RET

GETCH:    CALR LNENT
          CALR LFETCH; TESTB RL0; JR Z,GTER; CALR LINC
          CALR LNEX; RET
GTER:     SETFLG C; CALR LNEX; RET
BACKCH:   CALR LNENT
          CALR LDEC; CALR LSTORE
          CALR LNEX; RET

!Incrementa o decrementa il puntatore di linea!

LINC:     LD R2,LNPT; INC R2; CP R2,LNSZ; JR GE,LNER
          INC  LNPT; LD R1(#LPTR),LNPT; JR LNOK

LDEC:     TEST LNPT; JR MI,LNER
          DEC  LNPT; LD R1(#LPTR),LNPT; JR LNOK

!Preleva o memorizza per mezzo del puntatore di linea!

LSTORE:   TEST LNPT; JR MI,LNER
          LD R2,LNPT, ADD R2,#LDAT; LDB R1(R2),RL0; JR LNOK

LFETCH:   TEST LNPT; JR MI,LNER
          LD R2,LNPT; ADD R2,#LDAT; LDB RL0,R1(R2)
LNOK:     RESFLG C; RET
LNER:     SETFLG C; RET

!Commutazione del contesto all'ingresso o all'uscita!

LNENT:    EX LNRT,@SR; DEC SR,#LNRGB; LDM @SR,R2,#NLNRG
          LDM LCON,@R1,#NLNCX; JP @LNRT

LNEX:     POP LNRT,@SR; LDM R2,@SR,#NLNRG; INC SR,#LNRGB
          EX LNRT,@SR; RET
```

Figura 8.9 — Routine LINE

```
!Routine di interazione con il terminale
  CALL SAY; R1 = indirizzo di una stringa terminata con zero
  CALL ASK; R1 = indirizzo della domanda
  La routine ritorna con la risposta disponibile per l'elaborazione di GETCH e
  BACKCH
!
SAY: PUSH @SR,R1            !Salva l'indirizzo di stringa!
     DEC SR,#LOUTAB         !Costruisce la tabella TYIO nello stack!
     LD SR(#PADR),R1        !ADR = indirizzo della stringa!
     LD R1,#OUTPUT          !FLAG = «uscita»!
     LD SR(#PFLAG),R1
     LD R1,#TYOUTP          !OUTA = indirizzo di I/O!
     LD SR(#POUTA),R1
     LD R1,SR; CALL TYIO    !Inizia le operazioni di uscita!
     INC SR,#LOUTAB         !Ripristina lo stack!
     POP R1,@SR; RET        !Ripristina l'indirizzo di stringa; uscita da routine!

ASK: CALR SAY               !Invia la domanda in uscita!
     PUSH @SR,R1            !Salva l'indirizzo della domanda!
     DEC SR,#LINTAB         !Costruisce la tabella TYIO nello stack!
     LD R1,#INPUT           !FLAG = «ingresso»!
     LD SR(#PFLAG),R1
     LDA R1,INLINE          !ADR = indirizzo di INLINE!
     LD SR(#PADR),R1
     LD R1,#TYOUTP          !OUTA = indirizzo di I/O!
     LD SR(#POUTA),R1
     LDA R1,INRING          !RING = indirizzo di INRING!
     LD SR(#PRING),R1
     LDA R1,INLINE; CALL EMPLIN !Inizializza la linea di ingresso!
     LD R1,SR; CALL TYIO        !Inizializza l'ingresso!
     CALL TYWAIT                !Aspetta finché è fatto!
     INC SR,#LINTAB             !Libera la tabella dallo stack!
     LDA R1,INLINE; CALL SETLIN !Inizializza per GETCH, BACKCH!
     POP R1,@SR; RET            !Ripristina l'indirizzo della domanda;
                                uscita dalla routine!
!Un altro pezzo della routine di linea
SETLIN: PUSH @SR,R0; CLR R0     !Prende a prestito R0!
        LD R1(#LPTR),R0         !Punta al primo carattere!
        POP R0,@SR; RET         !Ripristina R0!
```

Figura 8.10 — ASK e SAY

In Figura 8.10 sono illustrate le routine SAY e ASK. Ciascuna di esse utilizza il suo argomento d'ingresso per predisporre una tabella di argomenti per la chiamata di TYIO. Si assuma che i buffer INLINE e INRING e l'indirizzo di I/O di TYOUTP siano definiti da qualche altra parte. Nel Capitolo IX, quando si discuterà un sistema elementare di timesharing «pilotato da eventi», vedremo che questi elementi divengono argomenti impliciti di SAY e ASK e variano da utente a utente.

In Figura 8.2 vengono definiti i simboli LINTAB, LOUTAB, PFLAG, ecc. Le tabelle degli argomenti per SAY ed ASK possono essere tenute nello stack e rilasciate appena viene eseguito il ritorno da TYIO, siccome l'informazione necessaria per supportare l'operazione in sviluppo iniziata da TYIO viene prelevata dalla tabella di TYIO e predisposta in CONBLK. (Naturalmente, questo richiede che le memorie dati e stack siano coincidenti). D'altra parte, gli elementi, come le stringhe, che devono essere inviate in uscita non possono essere lasciati nello stack, siccome CONBLK contiene solamente un puntatore alla stringa, ma non la stringa. Affinché una stringa, del tipo individuato da R1 per SAY, sia memorizzata nello stack, occorre trovare un modo per evitare il rilascio dello spazio di stack fino a quando è completata la stampa corrente. Si potrebbe avere in alternativa una linea chiamata OUTLIN, con funzione simile agli altri buffer fissi INLINE e INRING; si potrebbe quindi implementare una funzione SAY che trasferisce la stringa in OUTLIN e chiama TYIO con ADR puntato alla parte di testo di OUTLIN.

ESERCIZIO 7: Scrivete una routine, chiamata SAYF, che si comporti come suggerito nel precedente paragrafo. Qual è il modo migliore per avere l'indirizzo della parte di testo di OUTLIN? Qual è il modo più efficiente per trasferire la stringa in OUTLIN? Come potete assicurarvi che OUTLIN sia libero prima di trasferire la stringa dentro di esso?

Decodifica dell'Ingresso

Le routine GETCH e BACKCH possono essere utilizzate per implementare un certo numero di routine di decodifica dell'ingresso. Siccome INLINE viene usata in modo implicito da ASK, fa senso avere una coppia di routine GETC e BACKC che fa la stessa cosa di GETCH e BACKCH. Queste sono riportate in fondo alla Figura 8.11.

La Figura 8.11 è una illustrazione del tipo di elaborazione di una linea di ingresso, spesso utile nella programmazione interattiva. Il programmatore potrebbe usare una chiamata ad ASK per visualizzare il messaggio di «prompt» (pronto) ed aspettare l'ingresso del comando. I «campi» della

risposta si possono ottenere, usando GETFLD, uno alla volta. Per esempio, se la linea di ingresso è

LOAD «Game Program», 1, FF00

la successiva chiamata di GETFLD fornirà delle stringhe di caratteri ASCII terminate con zeri per

LOAD

«Game Program»

1

FF00

Onde riconoscere un particolare tipo di argomento, le chiamate di GETFLD dovrebbero essere intercalate con chiamate di routine «di riconoscimento» sviluppate per esaminare una stringa terminata con zero. Nell'esempio sopra riportato potremmo avere un riconoscitore di comando, un identificatore dei nomi del file, un identificatore del numero dell'unità disco ed un riconoscitore dell'indirizzo di memoria. La Figura 8.12 mostra come potrebbe essere fatta la codifica. Notate come sia ripetitiva. In effetti, questo tipo di applicazione è l'ideale per un approccio *basato su tabelle*: un insieme di quattro punti di ingresso a due elementi la descrive completamente. Perciò, si potrebbe scrivere un programma generale che faccia la stessa cosa elaborando una tabella della forma

RECCMD;CMD; RECFIL;FILCOD;
RECDSK;UNIT; RECADR;MEMADR

ESERCIZIO 8: Scrivete una versione del codice di Figura 8.12 che prenda l'indirizzo della tabella sopra riportata e fornisca le stesse cose che dà quella di Figura 8.12. Come potreste modificare il vostro programma in modo che sia possibile specificare un numero variabile di campi?

La maggior parte della regolarità della Figura 8.12 è dovuta all'uniformità delle convenzioni utilizzate per le routine RECCMD, RECFIL, RECDSK e RECADR: ciascuna preleva argomenti nel formato con cui lo restituisce GETFLD; questa routine restituisce C = 0 ed un codice di una parola per l'elemento riconosciuto in R0, oppure restituisce C = 1 se la stringa non è nel formato giusto. La Figura 8.13 illustra un identificatore di comando che utilizza la tecnica delle tabelle TRAN in cascata. Si as-

```
!Preleva il «campo» successivo dalla linea di ingresso

    CALL GETFLD: R1 = indirizzo del buffer in cui memorizzare il campo
                 R2 = dimensioni del buffer (byte)
    La routine ritorna con RL0 = «spazi» che seguono il separatore e vengono sal-
    tati; non viene passato nulla di intatto di ciò che è compreso tra le virgolette
    (« »); R2 = byte non usati (neg = supero)
!
GETFLD:  PUSH @SR,R1; CLRB RH0          !Posiziona l'indicatore: non è
                                        tra le virgolette!
GETL:    CALL GETC                      !Prossimo carattere di ingresso!
           JR C,GETEND                  !C significa EOL!
         CPB RL0,#QUOTE                 !Se ci sono le virgolette, inverti
                                        l'indicatore!
           JR NE,GET1; COMB RH0
GET1:    TESTB RH0; JR NZ,GET2          !Se la stringa è tra virgolette,
                                        passala!
         CALL CKSEP; JR NC,GETEND       !Altrimenti controlla il separato-
                                        re!
GET2:    DEC R2; JR MI,GETL             !Vede se c'è spazio per memo-
                                        rizzarla!
       LDB @R1,RLO;INCR1;JR GETL        !Se c'è spazio la memorizza ed
                                        incrementa il puntatore!
GETEND:  DEC R2; JR PL, GET3; DEC R1
GET3:    CLRB @R1                       !Memorizza il terminatore zero!
         CALL SKBLNK                    !Salta gli spazi!
         POP R1,@SR; RET

CKSEP:   PUSHL @ SR,RR0; CLRB RH0       !Salva R0, R1; prepara R0!
         LDA R1,SEPTAB; CALL TRAN       !È un carattere di separazione?!
         POPL RR0, @ SR                 !Ripristina R0, R1!
         RET                            !Passa il valore C di TRAN!
SEPTAB:  COMMA;0; SEMIC;0; SPACE;0; TAB;0; 0;0

SKBLNK:  PUSHL   SR,RR0                 !Salva R0, R1!
         CLRB RH0                       !Prepara per TRAN!
SKL:     CALL GETC; JR C,SKEX           !Prende il carattere successivo!
         LDA R1,WHITAB                  !Tabella degli spazi!
         PUSH @ SR,R0; CALL TRAN        !È uno spazio?!
         POP R0,@ SR; JR NC,SKL         !Se è così, prende il carattere
                                        successivo!
```

Figura 8.11 — Routine Campione per la Decodifica della Linea di Ingresso (segue)

```
        CALL BACKC                      !Se non è così, lo rimette a po-
                                        sto!
SKEX:   POPL RR0,@SR; RET
WHITAB: SPACE;0; TAB;0; 0;0

GETC:   PUSH @ SR,R1; LDA R1,INLINE; CALL GETCH
        POP R1,@ SR; RET

BACKC:  PUSH @ SR,R1; LDA R1,INLINE; CALL BACKCH
        POP R1,@ SR; RET
```

Figura 8.11 — Routine Campione per la Decodifica della Linea di Ingresso

```
!Codifica di un esempio di elaborazione di un ingresso!

        DEC SR,#BUFL; LD R3,SR          !Fai il buffer sullo stack!
REASK:  LDA R1,PROMPT; CALL ASK         !Invia il comando per l'ingresso
                                        di una nuova linea!
        LD R1,R3;LD R2,#BUFL            !1° campo: codice comando!
        CALL GETFLD
        CALL RECCMD; JR C,REASK
        LD CMD,R0

        LD R1,R3; LD R2,#BUFL           !2° campo: codice del filo!
        CALL GETFLD
        CALL RECFIL; JR C,REASK
        LD FILCOD,R0

        LD R1,R3; LD R2,#BUFL           !3° campo: numero del disco!
        CALL GETFLD
        CALL RECDSK; JR C,REASK
        LD UNIT,R0

        LD R1,R3; LD R2#BUFL            !4° campo: indirizzo di memoria!
        CALL GETFLD
        CALL RECADR; JR C,REASK
        LD MEMADR,R0

        INC SR,#BUFL                    !Rilascia il buffer di stack!
```

Figura 8.12 — Esempio di Codice per l'Elaborazione dell'Ingresso

sume che i simboli della tabella in fondo siano definiti da qualche altra parte. AS_E, AS_L, hanno lo stesso valore dei codici ASCII di E, L, ecc. C_EDIT, C_LIST, ecc. sono piccoli numeri distinti che riconoscono i comandi, esempio C_EDIT = 1; C_LIST = 2; ecc. Questi possono essere utilizzati più avanti dalle Figure 6.12 e 6.13 per il trasferimento alla corretta routine di elaborazione in una tabella TRAN simile a TYGO e GHRGO. NXCH è una routine utilizzata per prelevare dei caratteri dal buffer puntato da R1.

Il materiale riportato in Figura 8.11, 8.12 e 8.13 è un esempio del genere di strumenti che si possono mettere insieme per la gestione dell'ingresso; è possibile fare molto di più di quello che è mostrato qui.

ESERCIZIO 9: (a) Scrivete la routine NXCH chiamata in Figura 8.13. Si dovrebbe comportare come segue:

Se R1 = −1, chiama GETC e ritorna
Altrimenti LDB RL0,@R1; se RL0 poi INC R1 e ritorna
C = 0; se RL0 = 0, ritorna C = 1.

Questa routine è stata pensata per fare un ciclo che, qualunque sia il codice che noi scriviamo per elaborare le stringhe nel buffer, possa essere usato anche per elaborare la stringa in ingresso in INLINE.

(b) Se si aggiunge il comando ERASE, come dovrebbero cambiare le tabelle di Figura 8.13?

(c) Notate come dobbiamo azzerare spesso lo spazio RH0 prima di usare TRAN per trasmettere un carattere letto da GETC, GETCH, NXCH, ecc. Sarebbero possibili alcuni cambiamenti: fornire una versione di TRAN che converte solamente i caratteri che guardano ad RL0 (ma restituisce in R0 l'intera parola tradotta); avete GETCH, ecc., azzerate RH0. Quali sono i pro ed i contro di questi cambiamenti? Esistono altre possibilità? Quali effetti avrebbero questi cambiamenti sul codice già scritto?

(d) Scrivete un identificatore di comando che restituisca RH0 = mese, RL0 = giorno; R1 = anno. Che cosa dovrebbe essere in grado di riconoscere? Si può utilizzare una cascata di tabella TRAN per riconoscere i nomi dei mesi? I numeri romani per i mesi?

(e) Scrivete un riconoscitore per un ingresso numerico: riconosce i numeri esadecimali, decimali, ottali e binari con quattro riconoscitori separati che condividono routine comuni. Ciascuna dovrebbe essere in grado di trattare con un segno meno o più iniziale.

(f) Scrivete un riconoscitore per un ingresso temporale. Accetta le quattro cifre dell'orologio militare o il formato abituale dell'orologio (xx:xx AM, PM, M, N). Assicuratevi che il formato abituale funzioni senza uno zero quando le ore sono espresse da una sola cifra.

Formattazione dell'uscita

Mentre gestiamo l'ingresso, il maggior problema consiste nel comprendere ciò che viene digitato, il problema più importante per l'uscita è la formattazione. Esistono due strumenti utilizzabili per quello che discuteremo

!Riconoscitore di comando!

CALL RECCMD; I registri sono predisposti da GETFLD

La routine ritorna con C = 0 ed il codice cmd in R0
C = 1 se il comando non è riconosciuto

Il programma analizza uno alla volta i caratteri di ingresso: ad ogni punto, se ciò che è stato visto definisce in modo univoco un comando o probabilmente non può essere un comando, allora viene eseguito un opportuno ritorno alla routine; altrimenti il programma continua ad esaminare i caratteri. Questo è realizzato con una cascata di tabelle di TRAN: la prima converte ciascun possibile carattere iniziale sia in un codice comando sia nell'indirizzo di un'altra tabella TRAN di possibili secondi caratteri per quel carattere iniziale, e così via. I Codici Comando sono distinti dagli indirizzi essendo rappresentato da un codice 2* codice + 1; gli indirizzi non sono mai dispari.

```
RECCMD:  CLR R0; PUSH @SR,R1          !Inizializza!
         LDA R1,TBLZER
RCMDLP:  EX R1,@SR; CALR NXCH         !Carattere successivo — sbaglia
                                      se non c'è!
         EX R1,@SR; JR C,RCMDER
         CALL TRAN; JR C,RCMDER       !Converte — sbaglia se non può!
         RESFLG C
         RRC R0; JR C,RCMDOK          !Se codice, esce dalla routine!
         RLC R0                       !Altrimenti guarda la tabella
                                      TRAN per il successivo!
         LD R1,R0; CLR R0; JR RCMDLP
RCMDOK:  RESFLG C; POP R1,@SR; RET
RCMDER:  SETFLG C; POP R1,@SR; RET

TBLZER:  AS__E; 2*C EDIT + 1; AS__L; TB__L; AS__P; TB__P
         AS__T; TB__T; 0;0

TB__L:   AS__I; 2*C__LIST + 1; AS__O; 2*C__LOAD + 1;0;0

TB__P:   AS__R; TB__PR; 0;0

TB__PR:  AS__I; 2*C__PRINT + 1; AS__O; 2*C__PROGRAM + 1;0;0

TB__T:   AS__E; 2*CTEST + 1; AS__I; 2*C__TIME + 1
         AS__Y; 2*C__TYPE + 1; 0;0
```

!Le tabelle riportate sopra riconoscono i comandi della lista seguente: **EDIT, LIST, LOAD, PRINT, PROGRAM, TEST, TIME, TYPE!**

Figura 8.13 — Identificatore di Comando

!Routine per la formazione della linea

CALL SLIN; Aspetta fino a quando è fatta l'ultima operazione di uscita, poi posiziona OUTLIN.

CALL ELIN; Termina la linea (con zero), poi chiama SAY per inviare in uscita OUTLIN.

CALL ADDCHR; RL0 = carattere; Aggiunge il carattere a OUTLIN.

CALL ADDSTR; R1 = indirizzo della stringa terminata con zero; lo aggiunge a OUTLIN.

```
!
SLIN:     CALL TYWAIT                      !Aspetta le operazioni prece-
                                           denti!
          PUSH @ SR,R1                     !Salva R1!
          LDA R1,OUTLIN; CALL EMPLIN       !Inizializza la linea!
          POP R1,@ SR; RET                 !Ripristina R1 ed esce dalla
                                           routine!

ELIN:     PUSHL @ SR,RR0                   !Salva R0,R1!
          CLR R0; CALL ADDCHR              !Termina la linea!
          LDA R1,OUTLIN; ADD R1,#LDAT      !Punta al testo!
          CALL SAY                         !Lo invia in uscita!
          POPL RR0,@ SR; RET               !Ripristina R0,R1 ed esce
                                           dalla routine!

ADDCHR:   PUSH @SR,R1                      !Salva R1!
          LDA R1,OUTLIN; CALL ADLINE       !Aggiunge il carattere!
          POP R1, @SR; RET                 !Ripristina R1; ed esce dalla
                                           routine!

ADDSTR:   PUSH @ SR,R1; PUSH @ SR,R2       !Salva R1,R2!
          LD R2,R1; LDA R1,OUTLIN          !Aggiunge la stringa!
          CALL ADLNST
          POP R2,@ SR; POP R1,@ SR         !Ripristina R1,R2 ed esce
                                           dalla routine!
          RET
```

Figura 8.14 — Alcune Routine per la Formazione della Linea

qui di seguito: il controllo del cursore e la formazione della linea. Incominceremo con la formazione della linea, siccome ci fornisce una soluzione concreta al problema evidenziato nell'Esercizio 6: dove risiede la stringa mentre viene inviata in uscita? Il programma di base per la formazione della linea è costituito da una linea (chiamata OUTLIN) e da alcune routine:

SLIN: inizia la linea (aspetta fino a quando è libera la linea)

ELIN: termina la linea (chiama SAY per inviarla in uscita)

ADDxxx: un insieme di routine per aggiungere cose alla linea, esempio, ADDSTR aggiunge una stringa terminata con zero, ADDDEC aggiunge i caratteri ASCII per la rappresentazione decimale di un certo numero, ADDCHR aggiunge un singolo carattere, ADDDAT aggiunge una data, e così via.

La Figura 8.14 mostra le routine di base SLIN, ELIN e ADDCHR. Probabilmente si può costruire tutto partendo dalla routine ADDxxx ed utilizzando chiamate ad ADDCHR, ma questo non è funzionale. Per questa ragione, è stata aggiunta la routine ADLNST alle routine per la gestione della linea di Figure 8.8 e 8.9. La routine ADDSTR di Figura 8.14 utilizza ADLNST. ADLNST è presentata in Figura 8.15 insieme con EMPLIN, che viene chiamata da SLIN.

ESERCIZIO 10: (a) Notate che SLIN chiama TYWAIT; sebbene ci siano altre cose che TYOUT potrebbe fare oltre all'invio in uscita da parte di OUTLIN (per esempio l'eco). C'è un modo migliore per assicurarsi che OUTLIN sia libera prima di indirizzarla?

(b) Considerate lo schema seguente: a TYOUT viene aggiunto un altro stato corrispondente a FLAG = BUFOUT. In questo caso, TYOUT prenderà il suo prossimo carattere da inviare in uscita da un buffer ad anello; se l'anello è vuoto, allora TYOUT entra in uno stato di attesa corrispondente a FLAG = BSLEEP. Nel frattempo, ogni volta che ELIN viene chiamata trasferisce i contenuti di OUTLIN nel buffer ad anello. Essa non ritorna al chiamante fino a quando OUTLIN è libera, per cui SLIN non deve mai eseguire un controllo. Se ELIN trova FLAG = BSLEEP, mentre sta aggiungendo dei caratteri, pone FLAG = BUFOUT ed esegue un SC #IOGO per fare ripetere ogni cosa.

Trovate i dettagli di questo schema.

(c) Scrivete la routine ADDDAT che aggiunge una data a OUTLIN. Essa dovrebbe avere i suoi argomenti espressi nel modo seguente: RH0 = mese (1-12); RL0 = giorno (1-31); R1 = anno; R2 = codice formato. Tra le opzioni che il chiamante dovrebbe essere in grado di specificare ci sono:

Mese: nome intero, abbreviazione, numero
Anno: 4 cifre, ultime due cifre
Giorno, numero, numero seguito dal suffisso (st, nd, rd, th)
Ordine: mm/gg/aa; gg/mm/aa; mese giorno anno; giorno mese anno.

Inoltre, il programma dovrebbe essere in grado di eseguire dei calcoli e fornire in vari formati il giorno della settimana.

(d) Ora riscrivete ADDDAT per prelevare la data o il codice formato da una tabella puntata da R1. Perchè è migliore? Modificatela ancora in modo che la tabella contenga l'*indirizzo* della data. È ancora migliore?

Controllo Cursore

Le routine di formazione linea ci permettono di discutere il controllo del cursore.

La maggior parte dei videi permette un qualche controllo, da calcolatore, della posizione del cursore. Tra i costruttori non c'è molta standardizzazione, in molti casi il comando per inviare il cursore alla riga L colonna C è una sequenza di caratteri del tipo:

C1, C2, B1 + L, B2 + C

costituita da due caratteri di controllo (codici ASCII da 0 a 1F) seguita dai numeri reali della colonna e della riga, possibilmente con una costante di spiazzamento. La Figura 8.16 illustra come realizzare una routine ADDPOS per aggiungere una stringa opportuna ad OUTLIN. CURPOS chiude ADDPOS tra le chiamate SLIN ed ELIN. SCREEN mostra come visualizzare una videata composta di stringhe poste in determinate posizioni di riga e colonna. SCREEN è molto utile per visualizzare informazioni sullo schermo, essendo totalmente basata su tabella.

ESERCIZIO 11: (a) Scrivete una versione di SCREEN che utilizzi ADDPOS e ADDSTR piuttosto che CURPOS e SAY. Come fate a sapere quando dovete eseguire le chiamate di SLIN ed ELIN? È importante sapere quanto diventi pieno OUTLIN prima di chiamare ELIN?

(b) Modificate SCREEN in modo che essa riconosca altri comandi speciali. Per esempio, se la posizione della parola è −2, fate in modo che l'ingresso sia costituito da altri due elementi: un codice posizione seguito dall'indirizzo di una tabella argomenti per ADDDAT. Se le routine ADDxxx sono tutte scritte in modo che esse prendano un singolo argomento (possibilmente un indirizzo della tabella) in R1, poi la corrispondenza tra i codici di posizioni speciali con le routine ADDxxx può essere incorporata in una tabella TRAN; esempio:

−2; ADDDAT; −3; ADDHEX; −4; ADDDEC;

```
!Routine addizionali di linea!

  CALL ADLNST; R1 = indirizzo della linea
               R2 = indirizzo della stringa terminata con zero.
  La routine ritorna con C = 0 se alla linea è stata aggiunta tutta la stringa
                         C = 1 se parte o tutta non è stata aggiunta.

  CALL EMPLIN; R1 = indirizzo della linea
  Posiziona la linea a «vuota»
!
ADLNST: PUSHL @SR,RR2                  !Salva i registri!
        PUSHL @SR,RR0

        LD R0,@R1; LD R3,R1(#LPTR)     !Prende la dimensione (n) e il pun-
                                       tatore!
        DEC R0; SUB R0,R3              !Spazi lasciati (byte)!
          JR LE,ALSER                  !Insufficiente!
        ADD R1,#LDAT; ADD R1,R3
        INC R1
ALSLP:  TESTB @R2; JR Z,ALSOK          !Di più nella stringa?!
        INC R3; LDIB @R1,@R2,R0        !Incrementa il puntatore; muove
          JR NOV,ALSLP                 byte!
        TESTB @R2; JR Z,ALSOK          !Pieno – OK se la stringa è vuota!
ALSER:  SETFLG C; JR ALSOUT            !Niente spazio; C = 0!

ALSOK:  RESFLG C                           !All fit; C = 0!
ALSOUT: POPL RR0,@SR; LD R1(#LPTR),R3 !Aggiorna il puntatore!
        POPL RR2,@SR; RET

EMPLIN: PUSH @SR,R0; LD R0,#-1         !Prende a prestito R0!
        LD R1(#LPTR),R0                !Posiziona la linea a vuota!
        POP R0,@SR; RET                !Ritorna R0!
```

Figura 8.15 — Aggiunte alle Routine di Linea

(c) Scrivete una versione di ASK che prelevi in R0 l'argomento della posizione nello schermo e formuli la domanda alla colonna e linea specificate. Per la risposta dovrebbe avere la possibilità di mettere degli spazi vuoti in alcune posizioni del video? Come gestite le situazioni di Figura 8.12, se una risposta sbagliata fa ripetere la domanda al programma?

(d) La maggior parte dei videi ha un comando di pulizia schermo, generalmente realizzato inviando uno o due caratteri di controllo al dispositivo di uscita. Scrivete la routine di pulizia schermo ADDCLR e CLRSCR a-

!Posizionamento del cursore

CALL ADDPOS; RH0 = linea (da 0 a n − 1)
RL0 = colonna (da 0 a m − 1)

Ad OUTLIN vengono aggiunte la stringa di caratteri C1,C2,B1 + linea, B2 + colonna.

CALL CURPOS; R0 come in ADDPOS; viene posizionato il cursone.

CALL SCREEN; 31 = indirizzo della tabella delle posizioni ed indirizzi della stringa.

Formato della tabella:

POSITION (come sopra); STRING ADR; POSITION; ADR;...; − 1

```
!
ADDPOS:  PUSHL @ SR,RR0                  !Salva R0,R1!
         PUSH @ SR,#0                    !Forma la stringa nello
                                         stack!
         ADDB RH0,#B1; ADDB RL0,#B2
         PUSH @ SR,R0
         LDB RH0,#C1; LDB RL0,#C2
         PUSH @ SR,R0
         LD R1,SR; CALL ADDSTR           !Somma ad OUTLIN!
         INC SR,#6                       !Azzera lo stack!
         POPL RR0,@ SR                   !Ripristina R0,R1!
         RET

CURPOS:  CALL SLIN; CALL ADDPOS; CALL ELIN; RET

SCREEN:  PUSHL@SR,RR0; PUSH@SR,R2        !Salva R0,R1,R2!
         LD R2,R1                        !Il puntatore della tabella in
                                         R2!
SCREL:   CP @ R2,#−1; JR EQ,SCREX        !Fine della tabella?!
         LD R0,@ R2; LD R1,R2 (#2)       !Prende la posizione e l'indi-
                                         rizzo della stringa!
         INC R2,#4                       !Incrementa il puntatore!
         CALL CURPOS; CALL SAY           !Muove il cursore, chiama
                                         SAY!
         JR SCREL

SCREX:   POP R2,@ SR; POPL RR0,@ SR      !Ripristina R0,R1?R2!
         RET
```

Figura 8.16 — Programmi di Controllo Cursore

naloghe ad ADDPOS e CURPOS. Pensate di poter avere un nome migliore di CLRSCR?

La routine di video dovrebbe chiamare una routine di pulizia schermo prima di iniziare? Che cosa ne pensate di un codice speciale nella tabella di visualizzazione del video?

Come per l'elaborazione dell'ingresso, la nostra discussione della formattazione dell'uscita per il terminale non è andata più in là di una analisi superficiale. Abbiamo solamente discusso alcuni dei mezzi che renderanno più facile la maggior parte delle specifiche applicazioni.

!Programma per la gestione della tabella a bit

1. Gestione di bit e routine di test: **BSET, BCLR, BTST.**

   Chiama con R1 = indirizzo della tabella parametro; Restituisce C =1 se la posizione indicata è fuori dall'intervallo della tabella, altrimenti da C = 0. BSET e BCLR azzerano o mettono ad uno il bit; BTST pone Z ad uno se il bit è zero, altrimenti azzera Z.

2. Routine di Scansione della Tabella: **INBIT, NXCLR, NXSET, PVCLR, PVSET, NXBIT, PVIBIT.**

   Chiama con R1 = indirizzo della tabella parametro; per INIBIT, chiama con R0 = posizione da inizializzare (0 fino alla lunghezza – 1 della tabella in bit); restituisce C = 1 se la posizione data è esterna all'intervallo; altrimenti C = 0 e la posizione data diviene la posizione «corrente».

   NXCLR scandisce in avanti partendo dalla posizione corrente fino alla successiva con un bit a zero e la definisce come la nuova posizione «corrente»; C = 1 se si raggiunge la fine della tabella prima di trovare un altro bit a zero.

   NXSET scandisce in avanti alla ricerca di un bit ad uno; PVCLR e PVSET sono le routine corrispondenti in senso opposto. NXBIT e PVBIT eseguono semplicemente un passo alla volta.

3. Tutte le routine riportate sopra sono chiamate con R1 = indirizzo della tabella parametro. La tabella parametro ha il seguente formato:

   Prima parola: Indirizzo della tabella di bit

   ! **BTLEN** = **2** !2ª parola: lunghezza in bit della tabella!

   BTPOS = **4** !3ª parola: posizione corrente della tabella!

Figura 8.17 — Sequenza di Chiamata e Definizioni per un Programma di Tabella a Bit

Tabelle a Bit

Un'altra utile tecnica è la tabella a bit (esempio un insieme di indicatori costituiti da un solo bit ed impacchettati insieme in byte). Quando avete molti elementi e avete bisogno di conoscere solamente un unico fatto su di questi (al limite, come prima approssimazione), allora una tabella a bit costituisce un modo compatto, ed efficiente per gestire l'informazione. Per esempio, se aveste un file su disco di 10.000 blocchi, alcuni dei quali sono liberi per essere allocati e alcuni dei quali sono in uso. Una tabella a bit

```
!Routine di test e gestione!

BSET:     CALR BENT; JR C,BOUT
          SETB CURBY,BITNO; LDB @POINT,CURBY
BOUT:     CALR BEXIT; RET

BCLR:     CALR BENT; JR C,BOUT
          RESB CURBY,BITNO; LDB @POINT,CURBY; JR BOUT

BTST:     CALR BENT; JR C,BOUT
          BITB CURBY,BITNO; JR BOUT

!Routine di scansione!

INBIT:    LD R1(#BITPOS),R0; CALR BENT; JR BOUT

NXSET:    CALR NXBIT; JR C,NXER
          CALR BTST; RET C; RET NZ; JR NXSET

NXCLR:    CALR NXBIT; JR C,NXER
          CALR BTST; RET C; RET Z; JR NXCLR

PVSET:    CALR PVBIT; JR C,NXER
          CALR BTST; RET C; RET NZ; JR PVSET

PVCLR:    CALR PVBIT; JR C,NXER
          CALR BTST; RET C; RET Z; JR PVCLR

NXBIT:    LD R0,R1(#BITPOS); INC R0; JR Z,NXER
NXOK:     LD R1(#BITPOS),R0; RESFLG C; RET
NXER:     SETFLG C; RET

PVBIT:    LD R0,R1(#BITPOS); DEC R0; JR NOV,NXOK
          JR MI,NXER; JR NXOK
```

Figura 8.18 — Routine di Gestione della Tabella a Bit

Indicazione dell'entrata ed uscita della routine del programma di gestione della tabella di bit.

Routine di entrata: **BENT**

Salva tutti i registri da usare; se la posizione corrente è fuori dall'intervallo, restituisce C = 1, altrimenti C = 0 e posiziona i registri:

CURBY: Registro a byte contenente il byte reale della tabella di bit che contiene la posizione del bit corrente.

PARAM: Contiene l'indirizzo della tabella parametro.

POINT: Contiene l'indirizzo da cui è stato preso il byte in CURBY e a cui sarà inviata la versione modificata (per BSET, BCLR).

BITNO: Registro a parola contenente il numero del bit (da 0 a 7) della posizione del bit corrente in CURBY.

R0: Registro scratch.

Routine di uscita: **BEXIT**

Ricostituisce tutti i registri, conserva C,Z.

Figura 8.19 — Specificazione della Entrata e Uscita delle Routine del Programma nella Tabella a Bit

può fornire un accesso rapido a questa informazione con un minimo spazio di memorizzazione (in questo caso, 1250 byte).

Le Figure 8.17, 8.18, 8.19 presentano un programma per la gestione della tabella a bit. Le routine BENT e BEXIT, descritte in Figura 8.19, sono simili a tutte le altre routine di scambio di contesto che abbiamo discusso. Non è riportata la codifica reale.

Le routine BSET, BCLR e BTST sono ovviamente le routine di gestione della tabella. Le routine di scambio della tabella forniscono un utile sistema di elaborazione sequenziale.

ESERCIZIO 12: (a) Scrivete BENT e BEXIT.

(b) L'approccio seguito per le routine di scansione della tabella è molto chiaro ed immediato, ma richiede una notevole perdita di tempo. Calcolate il tempo usato da NXSET se viene posto ad uno il bit immediatamente successivo. Scrivete una versione di NXSET che elimini la perdita di tempo realizzando cose ad hoc; se possibile cercate di utilizzare le operazioni di blocco dello Z8000. Confrontate, in funzione della distanza dal bit successivo, il tempo richiesto dalla vostra versione con quella di Figura 8.18.

(c) Pensate a cinque applicazioni della tabella a bit. Quante di queste possono usare le routine di scansione?

(d) Con questa implementazione quale è la più grande tabella a bit utilizzabile? Quanti byte occuperebbe? Spiegate l'aritmetica dell'indirizzo usata in NXBIT e PUBIT e il controllo dell'errore.

Gli esempi sopra riportati forniscono una buona rappresentazione di come si possono costruire degli strumenti per realizzare dei programmi applicativi dello Z8000. Nel prossimo capitolo vedremo come si possa creare un ambiente di programmi applicativi.

Capitolo IX

Tecniche Avanzate di Programmazione

In questo capitolo verranno descritte un certo numero di metodologie che si applicano maggiormente ai sistemi nel loro insieme piuttosto che a specifici programmi applicativi.

Programmi Condivisibili e Gestione della Memoria

Nella maggior parte dei programmi che abbiamo presentato in questo libro, l'informazione viene elaborata nei registri non specializzati dello Z8000. Quando viene chiamata una subroutine, essa salva il contenuto dei registri che pensa di usare ponendoli nello stack; quando è pronta per eseguire il ritorno ricostituisce i valori dei registri prelevandoli dallo stack. In effetti, questa tecnica presenta alcuni problemi quando vogliamo usare la routine SAY, che fornisce, in interruzione, l'indirizzo di una stringa da inviare in uscita al programma di inizializzazione di I/O del terminale. Siccome, SAY esegue il ritorno verso il chiamante prima che sia completata l'uscita (rilasciando di conseguenza la sua area di stack), la stringa non potrebbe essere memorizzata da SAY nello stack.

Abbiamo precedentemente risolto i problemi di questo tipo utilizzando dei buffer del tipo INLINE e OUTLINE, ma non possiamo dire mai sicuramente *dove* siano questi buffer.

Se avete una certa esperienza con la programmazione dei minicalcolatori, potreste essere colpiti dal tipo di differenza esistente. Ci va bene che i buffer si trovino in un generico posto della memoria? Ci sono due possibili risposte a questa domanda, una relativa all'ambiente tipico in cui è inse-

rito il minicalcolatore e l'altra relativa alla possibilità di generalizzare ciò che abbiamo fatto fino ad ora.

La prima risposta è che desideriamo, considerata come possibile, memorizzare i nostri programmi nella memoria a sola lettura (ROM); la ROM è economica e fornisce una protezione contro modifiche inavvertite dei programmi. Per cui i buffer, le tabelle, gli stack e altre parti di memoria modificabili debbono essere attentamente separate dal programma. Questo materiale è generalmente memorizzato in memoria di lettura/scrittura (chiamata RAM – random access memory – (memoria ad accesso casuale) per ragioni puramente storiche). La RAM è generalmente volatile ossia il suo contenuto viene perso se viene tolta l'alimentazione. Naturalmente, anche i programmi possono essere memorizzati in RAM, come in effetti viene fatto generalmente durante la fase di sviluppo siccome le ROM, che potrebbero essere modificate utilizzando sistemi e circuiti speciali, sono di difficile uso nella parte di sviluppo del programma. Ma frequentemente gli obiettivi dei progetti basati su microprocessori sono i programmi memorizzati in ROM.

La seconda risposta è che i programmi che non hanno una propria memorizzazione locale, ma sono potenzialmente condivisibili, perciò essi possono lavorare simultaneamente con due diversi contesti. Un esempio di questo tipo è fornito dalle routine di linea di Figura 8.9: le routine LINC, LDEC, LFETCH ed LSTORE sono chiamate da routine di interruzione e non-interruzione. Per esempio, la routine GETFLD di Figura 8.11 chiama GETC; GETC chiama GETCH, GETCH chiama LFETCH e LINC. Supponete che mentre lo Z8000 sta eseguendo la LINC (chiamata da GETFLD per mezzo della catena di routine riportata sopra) ci sia una interruzione da parte del dispositivo terminale di uscita e nel corso dell'esecuzione di quella interruzione TYOUT chiami ADLINE, la quale chiama LINC.

Il contesto originale, su cui stava lavorando LINC (chiamata da GETFLD), è salvato nello stack dalla chiamata di TYOUT a TYENT. La chiamata di ADLINE a LNENT predispone un nuovo contesto su cui agisce LINC quando viene chiamata. Quando TYOUT chiama TYEXIT prima della sua IRET, LINE procede come se non si fosse assolutamente verificata una interruzione. Le istruzioni di LINC sono state usate contemporaneamente in due contesti diversi; perciò LINC è condivisibile.

Se LINC non fosse stata condivisibile, esempio memorizza le sue variabili in locazioni di memoria ad indirizzi fissi, allora avremmo dovuto avere una LINC per le routine ADLINE e BACKUP, dipendenti dalle interruzioni, ed una LINC diversa per GETCH e BACKCH. Attualmente, una qualsiasi delle routine ADLINE, BACKUP, GETCH, e BACKCH

può essere chiamata da un qualsiasi programma senza dovere duplicare le routine o una qualsiasi delle loro subroutine.

Timesharing (Condivisione temporale)

Esiste anche un uso più importante delle routine condivisibili: il time-sharing. Da una parte la suddivisione della memoria in routine condivisibili e dall'altra i dati che esse usano, costituiscono la base per una facile implementazione di un sistema di time-sharing. L'idea di base è la seguente: in molte applicazioni, specialmente quelle che richiedono una interazione tra gli operatori e il terminale, il calcolatore può gestire molte volte il caricamento richiestogli da un operatore. Per cui, invece di avere solo un insieme di dati per ogni routine condivisibile, potrebbe esserci un insieme comune per ogni terminale. Fra poco dovremo ritornare a questo argomento, ma per prima cosa guardiamo come possono essere organizzati i dati siccome l'idea di base è legata alla possibilità di identificare l'intero insieme di dati appartenenti ad un terminale, per poterli commutare rapidamente da un insieme ad un altro.

La maggior parte delle routine che abbiamo considerato utilizza intensamente lo stack per il salvataggio ed il ricostituimento delle routine. Se esse desiderassero una maggior capacità di memorizzazione di quella fornita dai registri, potrebbero usare l'area di stack come area lavoro, per cui lo stack sembra essere un buon punto di partenza. In effetti, nelle routine considerate nel Capitolo VI e Capitolo VIII, l'unica area di memorizzazione non appartenente allo stack erano i buffer INRING, OUTLIN, e INLINE e il blocco di contesto del terminale di I/O posto a CONBLK.

Parlando di time-sharing è utile introdurre un construtto che semplifichi le cose: *l'utente*. Questo non è riferito ad una particolare persona. Piuttosto è un termine associato ad un insieme di utilità del tipo stack, buffer ad anello e così via. Questo utente immaginario si dice faccia parte della prestazione che stiamo analizzando, noi parliamo dello stack dell'utente, del buffer ad anello dell'utente ecc.

Per cui, analizzeremo ora come sono organizzati lo stack e le INRING, OUTLINE, INLINE e CONBLK dell'utente. In Figura 9.1 è mostrata una delle possibili disposizioni, supponendo che tutti gli elementi, eccetto lo stack, siano di dimensione fissata. Questo raggruppamento in un blocco unico permette di calcolare tutti gli altri indirizzi partendo dall'indirizzo, in cima, a cui termina OUTLIN. Questo indirizzo e il puntatore di stack sarebbero sufficienti per specificare l'area dati dell'utente.

Però lo schema illustrato in Figura 9.1 non è flessibile (tutti gli utenti devono avere la stessa dimensione di buffer) perchè non considera il fatto che alcune informazioni devono essere accessibili nel momento in cui si

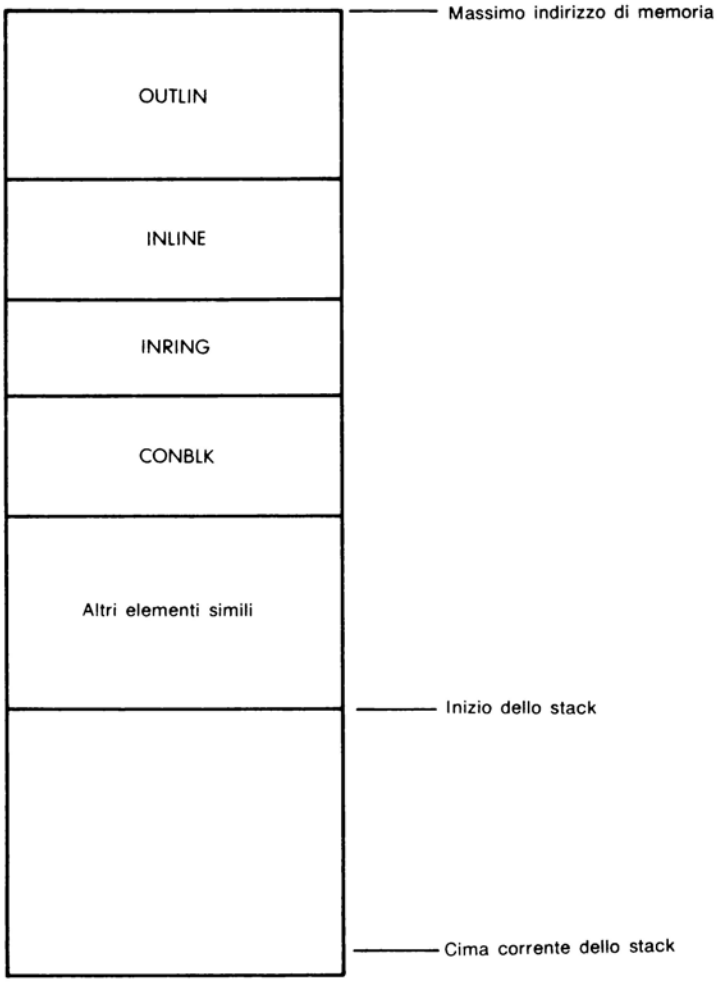


Figura 9.1 — Una Possibile Struttura dei Dati dell'Utente

verifica una interruzione (per cui, potrebbero essere raggruppate, in modo migliore, con altri elementi di questo tipo), mentre altre non sono necessarie. In Figura 9.2 viene presentato uno schema che soddisfa queste esigenze. Con questo schema, gli elementi possono essere ancora raggruppati nell'insieme superiore dell'area dati dell'utente, ma non è essenziale che siano allocati qui; possono trovarsi da qualsiasi altra parte.

Per gli esempi forniti nel resto di questo libro assumeremo una struttura simile a quella di Figura 9.2. Notate che se anche l'insieme di indirizzi e gli altri elementi simili non devono essere adiacenti allo stack dell'utente

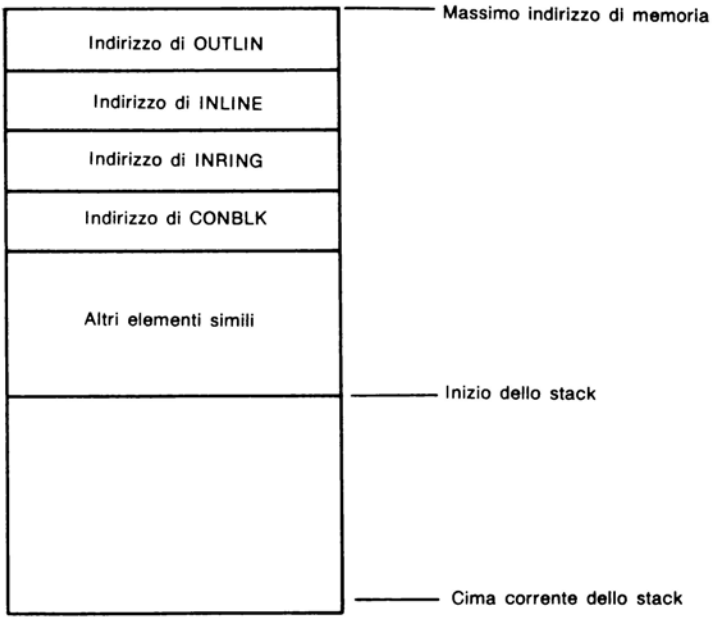


Figura 9.2 — Una Struttura Migliore

nella memoria, in realtà quest'area è un luogo conveniente dove metterli, specialmente se decidiamo in un secondo tempo di «spostare» i dati dell'utente dalla memoria centrale verso un dispositivo di memorizzazione secondario (esempio un disco). Questo insieme di locazioni fisse all'«altro estremo dello stack» è simile all'*heap* del PASCAL.

Naturalmente ora che ci sono molte copie di CONBLK, INLINE, ecc. occorrerà rivedere le routine che le utilizzano. Per fare questo, abbiamo bisogno di un modo per fare riferimento al dato specifico dell'utente. Una possibile soluzione consiste nel dedicare allo scopo un registro indirizzo, come appunto un registro indirizzo viene dedicato ad essere il registro dello stack. Quindi assumiamo che il registro HR punti, in Figura 9.2, all'indirizzo più alto dell'area dati dell'utente.

Allora si può accedere alla cella che contiene l'indirizzo di OUTLIN come @ HR o HR(#0); l'accesso alla cella che contiene l'indirizzo di INLINE viene individuato come HR(#−2) e così via. Notate che per permettere sia le operazioni segmentate che quelle non-segmentate, stiamo utilizzando il modo di indirizzamento tramite base (disponibile solamente con l'istruzione LD). Naturalmente, per le operazioni in versione segmentata, il registro indirizzo HR sarà una coppia di registri.

Guardiamo come dovremo modificare le diverse routine. Inizieremo con una facile: ASK (Figura 8.10). Notate che c'è due volte l'istruzione:

```
LDA R1,INLINE
```

ed una volta

```
LDA R1,INRING
```

Queste possono essere sostituite con

```
LD R1,HR(#INLINF)
```

e

```
LD R1,HR(#INRINF)
```

Queste sono basate sulle definizioni (Figura 9.2):

OUTLNF = 0; INLINF = −2; INRINF = −4; CONBLK = −6

In altre parole, dove la ASK originale preleva una INLINE ed una INRING, caricando i loro indirizzi con la LDA, adesso ce ne possono essere molte; ma quella particolare a cui siamo interessati ha il suo indirizzo in una locazione fissa dell'area dati dell'utente corrente. Se HR viene cambiato per puntare ad un'altra area utente, allora nella routine ASK la stessa codifica predisporrà l'INRING e INLINE del nuovo utente.

C'è un altro problema con ASK e SAY: esse trasmettono a TYIO l'indirizzo di I/O della stampante/video. Anche questo varia per ciascun utente, quindi in Figura 9.2 la parte indicata con «Altri elementi simili» preleva il suo primo elemento. La definizione

TYOTPF = −8

rimpiazzerà l'istruzione

```
LD SR(#POUTA),#TYOUTP
```

con

```
LD SR(#POUTA),HR(#TYOTPF)
```

Casualmente, avremo probabilmente bisogno di un elemento simile per l'indirizzo di I/O della tastiera, ma qui non lo useremo. Invece specifichiamo l'indirizzo dell'ingresso del buffer ad anello (HR(#INRINF)). La corrispondenza tra ciascun buffer ad anello ed il suo indirizzo di I/O della tastiera deve essere fatto da qualche parte nell'inizializzazione del sistema. Questa informazione viene registrata nel CONBLK dell'utente, essendo usata dalla routine TYIN.

ESERCIZIO 1: Con le linee di istruzione usate prima, modificate le routine GETC e BACKC di Figura 8.4. Fate i cambiamenti analoghi nelle routine SLIN, ELIN, ADDCHR e ADDSTR di Figura 8.14.

I successivi cambiamenti da fare si trovano nelle routine di interruzione TYOUT e TYIN, ma a causa del modo con cui sono organizzate, i cambiamenti vengono fatti in TYENT e TYEXIT. In questo caso il problema consiste nel decidere quale sia il CONBLK di utente da prelevare. Non è sufficiente fare semplicemente riferimento a HR(#CONBKF), siccome una volta che introduciamo il time-sharing, non è detto che il terminale dell'utente che sta interrompendo sia il programma dell'utente che è stato interrotto. Perciò, prima di avere l'indirizzo corretto da HR(#CONBKF) possiamo avere bisogno di cambiare HR.

Il problema consiste nel fatto che il registro HR che definisce l'area dati dell'utente, e per cui definisce argomenti impliciti alle routine SAY, ASK, GETC, BACKC, ecc., non è disponibile al momento dell'interruzione. Occorrerebbe trovare dei mezzi diversi per collegare la specifica interruzione ricevuta al blocco di contesto appropriato. Questo, a sua volta, dipende da come i terminali usano le prestazioni dello Z8000. Per esempio, supponiamo che ci siano otto terminali controllati da un dispositivo di commutazione comune alle prime due posizioni dell'interruzione vettoriale (Vedasi Figura 2.9); le interruzioni della tastiera sono collegate alla prima posizione vettoriale, mentre quelle del video alla seconda. Supponiamo anche che il byte più significativo della «ragione» posta nello stack dell'interruzione contenga un numero, compreso tra zero e sette, che specifica qual è il terminale che richiede attenzione.

Adesso supponiamo che ci sia una tabella di otto indirizzi memorizzata alla locazione CONTBL. Ciascuno di questi è l'indirizzo di uno dei blocchi di contesto e la tabella viene indicizzata mediante il numero associato al terminale. Assumiamo che TYENT venga chiamata mediante il numero associato al terminale posto in RH2. Questo causerà alcuni cambiamenti, che tratteremo fra poco, a TYIN, TYOUT e TYIO.

Adesso possiamo scrivere TYENT. Il blocco di contesto può essere e-

spresso per contenere un registro CONAD tra IOUT e TYRT. (Vedasi Figura 6.11). La linea

```
LDM TYCON,CONBLK,#NTYCX – 1
```

deve essere sostituita con

```
SRA R2,#8; LD CONAD,CONTBL (R2)
LDM TYCON,@CONAD,#NTYCX – 2
```

dove NTYCX ha il nuovo valore 8. Sia CONAD che TYRT non devono essere caricati dal blocco di contesto. Allora in TYEXIT la linea

```
LDM CONBLK,TYCON,#NTYCX
```

viene sostituita con

```
LDM @CONAD,TYCON,#NTYCX
```

I cambiamenti a TYOUT e TYIN richiedono, in due posti, l'aggiunta dell'istruzione

```
EX R2,@SR
```

La Figura 9.3 rappresenta una versione riveduta di TYIO. Notate che sono stati tolti l'indirizzo di I/O di uscita e l'indirizzo di anello. Questi (concettualmente) appartengono al terminale, non all'utente, per cui diventano punti fissi di entrata al blocco di contesto, predisposti all'inizializzazione del sistema. Invece, TYIO utilizza il numero associato al terminale (al quale fa riferimento per mezzo di HR(#TRMNOF)) per selezionare il corretto blocco di contesto del terminale.

ESERCIZIO 2: (a) Perché si usa R2 per trasmettere a TYENT il numero associato al terminale? Perché non R0 o R1? (Suggerimento: vedasi Figura 4.16 e Figura 9.3).

(b) Rivedete SAY e ASK in modo da farle funzionare con la nuova TYIO. Che cosa deve essere fatto alla routine TYWAIT della Figura 8.2?

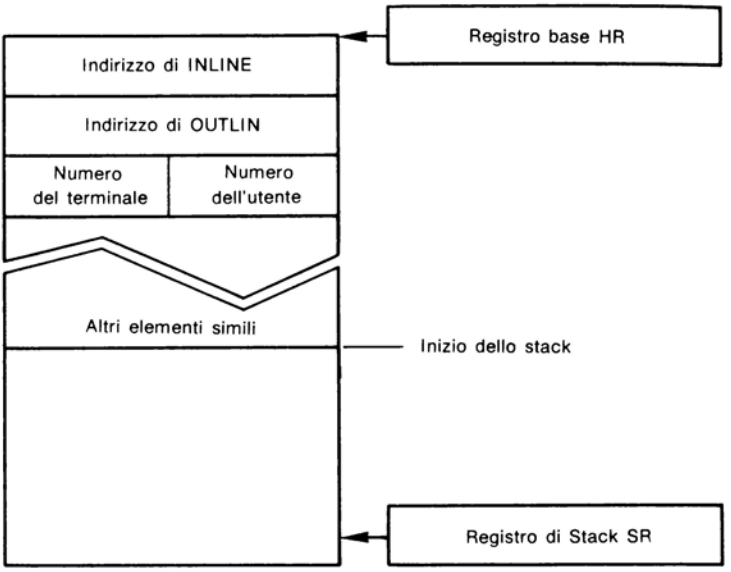
(c) Riscrivete le definizioni rivedute di TYENT e TYEXIT; riscrivete le nuove versioni di TYENT, TYEXIT, TYIN e TYOUT comprensive dei cambiamenti sopra descritti.

Per permetterci di introdurre il time-sharing riassumiamo ora ciò che è stato fatto per parametrizzare le routine dei Capitoli VI e VIII.

```
!Inizializzazione dell'I/O del terminale (versione non-segmentata)
  CALL TYIO; R1 = indirizzo della tabella parametro
  Ritorna alla routine con C = 0 se inizializza
                            C = 1 se il comando non è valido

  L'I/O lavora sotto interruzione.

  Formato della tabella parametro:
    Indirizzo Byte   Per Richiesta di Ingresso      Per Richiesta di Uscita

    PFLAG  = 0  !#INPUT                             #OUTPUT!
    PADR   = 2  !Indirizzo di LINE                  Indirizzo stringa uscita!
    LTYTAB = 4                  !Lunghezza della tabella!

TYIO:     CALR TYWAIT                              !Aspetta DONE!
          PUSH @SR,R2; LDB RH2,HR(#TRMNOF)!Numero del terminale!
          CALR TYENT                               !Posiziona il contesto!
          LDB FLAG,R1(#PFLAG+1)                    !Posiziona il FLAG &!
          LD PTR,R1(#PADR)                         !L'indirizzo tabella!
          CPB FLAG,#INPUT; JR EQ,TYOKX             !Controlla per!
          CPB FLAG,#OUTPUT; JR EQ,TYOKX            !Comando valido!
          LDB FLAG,#DONE                           !Cancella il comando!
          SETFLG C; CALR TYEXIT                    !Uscita per errore!
          POP R2,@SR; RET
TYOKX:    RESFLG C; CALR TYEXIT                    !OK esce dalla routine!
          POP R2,@SR; SC #IOGO; RET
```

Figura 9.3 — Revisione della Routine di Inizializzazione TYIO

- Abbiamo dato a ciascun utente un'area dati con un insieme di parametri posti ad un estremo, in posizione fissa relativamente al registro base HR, e con lo stack posto all'altro estremo. La Figura 9.4 mostra una versione comprendente gli ultimi cambiamenti.
- Abbiamo costruito una tabella di indirizzi del blocco di contesto di I/O del terminale indicizzati mediante il numero associato al terminale. Il numero associato al terminale diventa una parte dell'area dati dell'utente e gli elementi specifici del terminale come gli indirizzi di I/O e gli indirizzi di ingresso del buffer ad anello sono stati tolti dall'area utente e sono diventati elementi fissi nella tabella di contesto del terminale. (Vedasi Figura 9.5).
- Abbiamo fatto alcuni cambiamenti, relativamente piccoli, a



Definizione degli spiazzamenti degli elementi da HR:

INLINF	= 0	Indirizzo del buffer di ingresso di linea
OUTLNF	= −2	Indirizzo del buffer di uscita di linea
TRMNOF	= −4	Numero del terminale
USRNOF	= −5	Numero dell'utente

Figura 9.4 — Struttura dei Dati dell'Utente

TYENT, TYEXIT, TYOUT, TYIN e TYIO e alcuni cambiamenti, notevolmente inferiori, ad alcune altre routine.

Il time-sharing inizia in TYWAIT. La piccola routine in fondo alla Figura 8.2 è quella dove viene eseguita tutta l'attesa e, secondo il concetto fondamentale del time-sharing, il calcolatore dovrebbe eseguire il lavoro di un utente che ha qualche cosa da fare non appena quello corrente deve eseguire un'attesa. La Figura 9.6 riporta la versione di TYWAIT per realizzare il time-sharing. TYWAIT è ancora molto semplice per il modo con cui abbiamo organizzato la struttura. Ogni utente deve ricordarsi i contenuti dei 16 registri non specializzati. Quindici di questi sono memorizzati nell'area dati dell'utente e il sedicesimo, l'indirizzo dell'area dati, viene salvato in una tabella indicizzata dal numero dell'utente. Poi, partendo dal numero dell'utente posto dopo quello corrente, TYWAIT cicla, uno dopo l'altro, attraverso i vari utenti, controllando il blocco di contesto di cia-

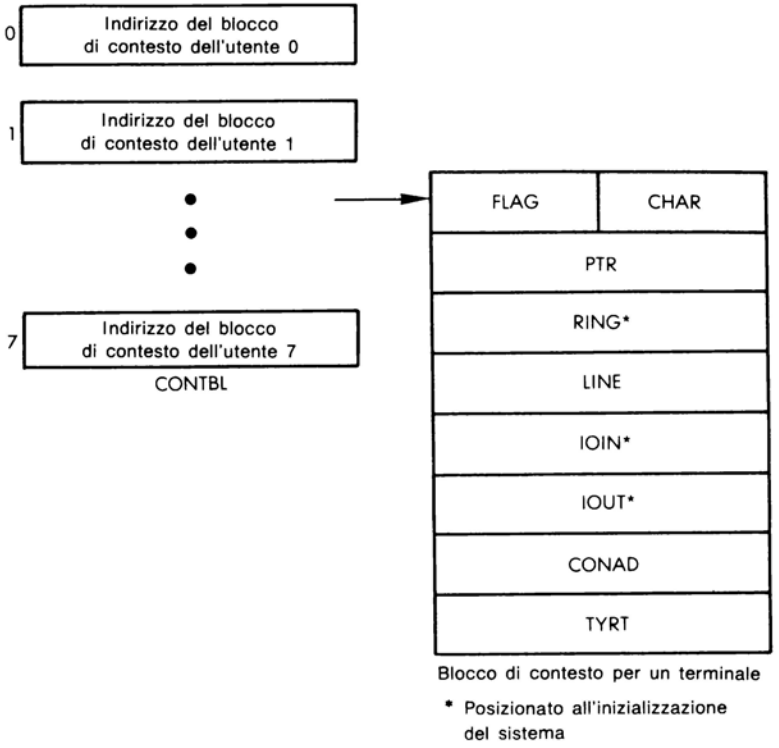


Figura 9.5 — Blocchi di Contesto del Terminale

scun terminale utente fino a quando non trova FLAG = DONE. Questo è un utente pronto per essere eseguito.

È semplice far partire un utente dopo un'attesa: è sufficiente non fare ciò che aveva fatto all'inizio di TYWAIT. Per mezzo di HR(#SRF) viene ricostituito il valore di SR, e gli altri 14 registri vengono ricaricati dallo stack. Infine, siccome la catena degli indirizzi di ritorno dalla subroutine è ancora nello stack utente, l'istruzione RET posta alla fine funziona giusto come se a nessun altro sia stato concesso di essere eseguito durante l'attesa. La Figura 9.7 mostra come venga salvato il tutto.

ESERCIZIO 3: (a) La parte dello «stato» che non viene salvato sopra è la parola di flag/controllo. Potrebbe essere salvata? Potrebbe essere salvata una parte di essa? Se voi desideraste salvarla, come fareste? (Suggerimento: ricordatevi LDCTL e LDCTLB).

(b) Che cosa succede se il valore permette agli utenti di aspettare che si verifichino per i loro terminali condizioni diverse da FLAG = DONE: esempio un utente può voler aspettare fino a quando sono trascorsi cinque secondi prima di cancellare un messaggio di errore dallo schermo. Che ge-

```
!Semplice Catalogatore/Scambiatore

CALL TYWAIT; ritorna dopo:
    1. Avere controllato ogni altro utente; e
    2. il blocco di contesto del vostro terminale ha FLAG = DONE
!
NWTRG = 14            !Suppone un funzionamento non-segmentato!
WTRGB = 2*NWTRG       !Per i registri occorrono dei byte della memoria di
                      stack!

!Usiamo la tabella HRTB, indicizzata dal numero dell'utente, per memorizzare il
valore di HR per ogni utente!

TYWAIT: DEC SR,#WTRGB                      !Salva i registri nello stack!
        LDM @ SR,R0,#NWTRG
        LD HR(#SRF),SR                     !Salva il puntatore di stack!

        LDB RL1,HR(#USRNOF)                !Preleva il # dell'utente cor-
                                           rente!
        CLRB RH1; LD HRTB(R1),HR           !Salva l'utente HR!
WTLP:   INC R1                             !Utente successivo!
        CP R1,#MXUSER; JR LT, WT1          !  (mod MXUSER)!
        SUB R1,#MXUSER
WT1:    LD HR,HRTB(R1)                     !Prende HR dell'utente!
        LD RL2,HR(#TRMNOF);CLRB RH2 !Poi il terminale dell'utente!
        LD R2,CONTBL(R2)                   !FLAG = DONE?!
        CPB @R2,#DONE
          JR NE,WTLP                       !No — non può essere esegui-
                                           to!
        LD SR,HR(#SRF)                     !Si — ricostituisci SR!
        LDM R0,@SR,#NWTRG                  !  e i registri!
        INC SR,#WTRGB
        RET                                !Infine ritorna!
```

Figura 9.6 — Una Semplice Lista di Circolazione per uno Scambiatore per Timesharing

nere di informazioni occorrerebbe fornire a TYWAIT? Dove dovrebbe essere presa?

(c) Notate che il valore di HRTB non cambia mai. Potrebbe essere fatto in modo diverso? Che cosa succederebbe se ci fossero più utenti che potrebbero avere allo stesso momento delle aree dati in memoria? Come potreste ricordarvi di quelli che erano in memoria e di quelli che non c'erano?

Come potreste fare per far sì che TYWAIT abbia sufficienti informazioni da poter essere in grado di controllare senza guardare nell'area dati dell'utente (che potrebbe non essere in memoria), se un utente è pronto per essere eseguito?

(d) Che cosa succede se un utente chiama TYWAIT quando il suo blocco di contesto del terminale ha FLAG = DONE? Come si potrebbe usare questa situazione?

Gestione dello Stack

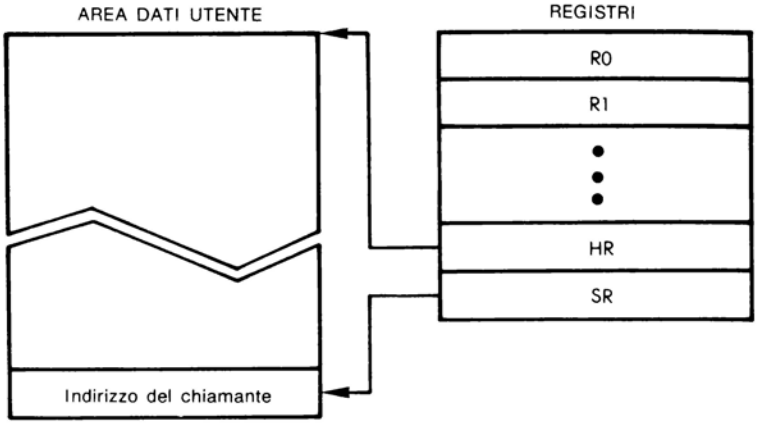
Per tutti i programmi e le configurazioni dei sistemi che abbiamo considerato, abbiamo visto come sia utile lo stack; ma anche l'uso degli stack presenta dei problemi:

- Gli indirizzi di ritorno sono sempre presi nello stesso modo; esempio guardate TYENT e TYEXIT.
- Una delle conseguenze, è l'estrema difficoltà nel passare degli argomenti tra le varie routine utilizzando lo stack.
- Ogni subroutine deve controllare l'uso del suo stack diligentemente in modo che essa rimuova esattamente tutto quello che aveva aggiunto; altrimenti la sua RET non funzionerà.

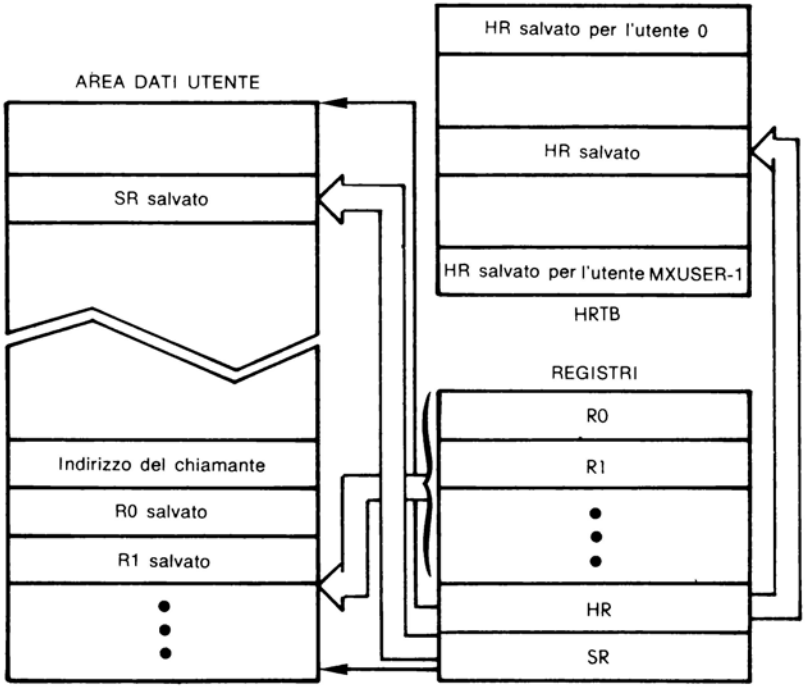
La soluzione a questi problemi è l'uso degli stack separati: uno stack utilizzato dal processore per le chiamate di subroutine ed un altro stack, gestito da software, per la memorizzazione temporanea e lo scambio degli argomenti tra routine.

La Figura 9.8 illustra come potrebbe funzionare questo meccanismo. Quando viene chiamata una subroutine, il suo registro dello stack di memorizzazione viene salvato nello stack chiamante insieme all'indirizzo di ritorno; poi all'uscita della routine viene ricostituito automaticamente lo stack di memorizzazione, per cui la routine X non ha bisogno di conservare traccia e rimuove lo stack di memorizzazione che aveva usato. Con questo meccanismo diventa immediato il passaggio di argomenti mediante stack: si può cambiare il valore salvato del puntatore dello stack di memorizzazione ed il nuovo valore viene ricostituito al ritorno. La Figura 9.9 mostra come potrebbe funzionare questo sistema.

Notate che abbiamo tenuto SR come nome del registro dello stack di memorizzazione ed abbiamo introdotto il nuovo nome SP per lo stack di chiamata. Questo significa che SP deve essere R15 per il funzionamento non-segmentato o RR14 per il funzionamento segmentato. SR può essere un qualsiasi registro indirizzo da noi scelto. Per concretezza, supponiamo che HR, SR e SP siano gli ultimi tre registri indirizzi: cioè R13, R14, R15 per il funzionamento non-segmentato oppure RR10, RR12, RR14 per quello segmentato.



STATO DELL'AREA UTENTE E DEI REGISTRI QUANDO È CHIAMATA TYWAIT



TUTTI I REGISTRI SALVATI DA TYWAIT

Figura 9.7 — Salvataggio dello Stato di un'Attesa Utente di I/O

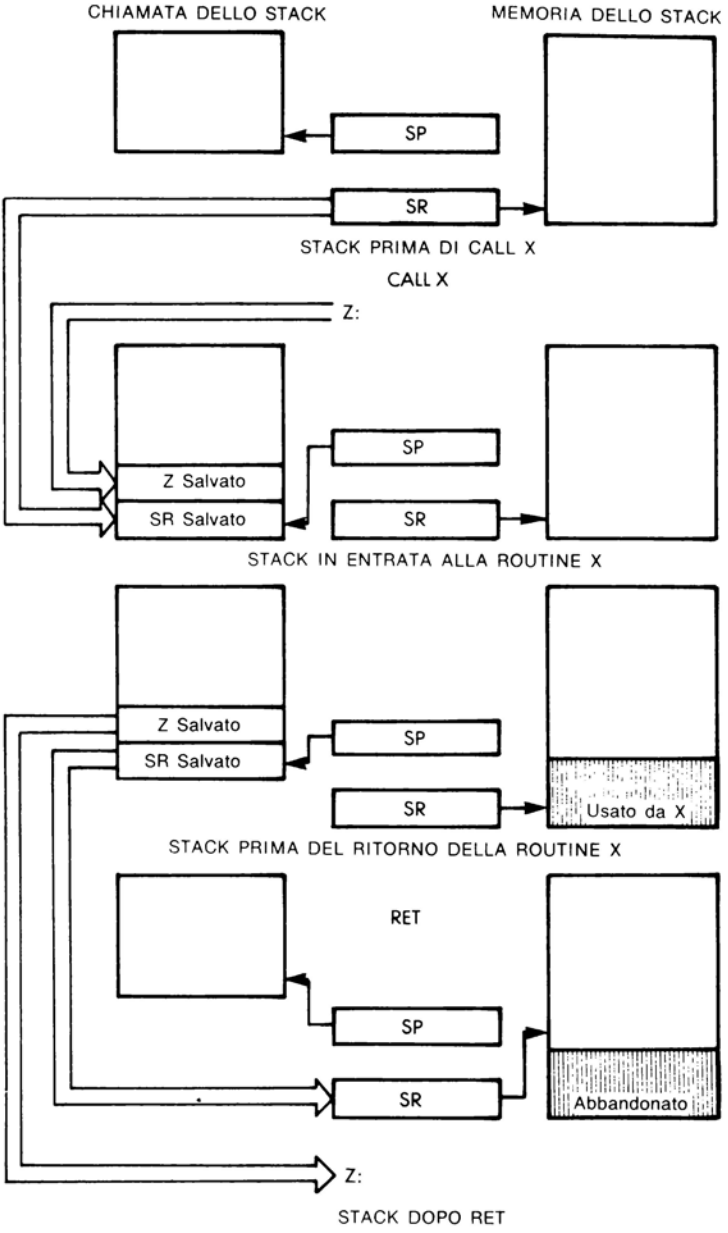


Figura 9.8 — Separazione degli Stack di Chiamata e di Memorizzazione

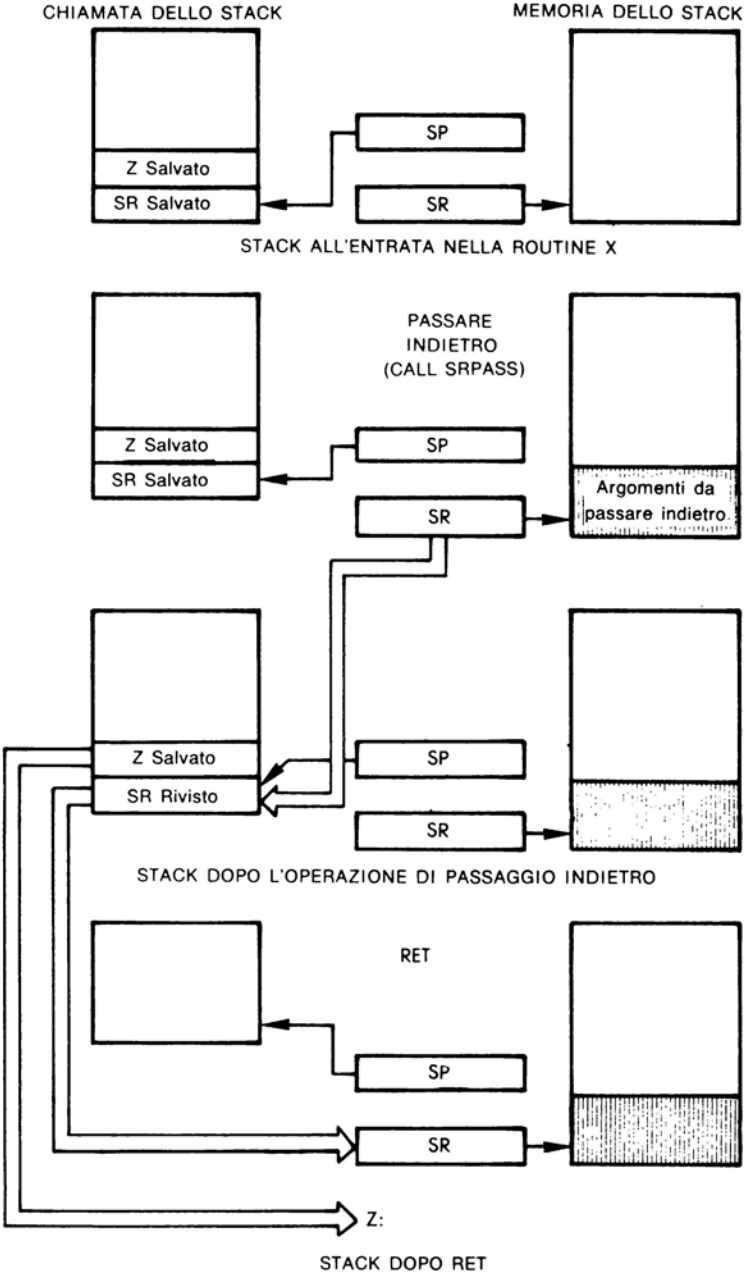
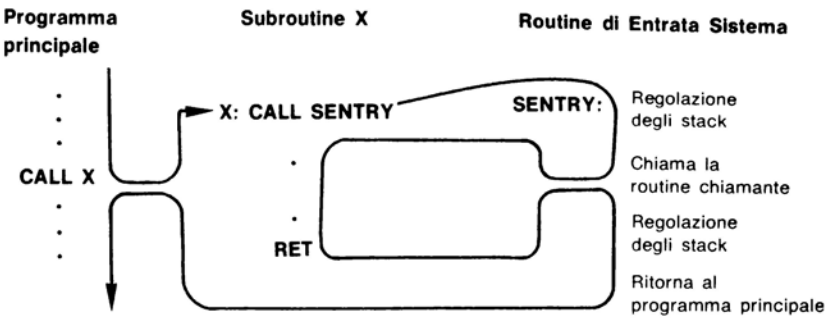


Figura 9.9 — Argomenti Inviati Indietro Tramite lo Stack di Memorizzazione



Le linee scure indicano il flusso di controllo; quindi l'ordine di esecuzione delle istruzioni.

Figura 9.10 — Flusso di Controllo quando si Usa un Programma di Entrata del Sistema

Non abbiamo ancora detto come fare funzionare tutto questo. Naturalmente, ciascuna routine potrebbe mettere SR nello stack SP all'entrata e riprenderlo prima di uscire — questo è un metodo poco attendibile e soggetto ad errore. Un approccio leggermente migliore, con alcuni piacevoli effetti collaterali, è costituito dal fatto che ciascuna routine, entrando, chiami una routine di sistema per implementare il salvataggio e il ricostituimento di SR. Poi la routine chiamata, chiama a sua volta il corpo della subroutine; per cui il RET della routine originale ritorna per mezzo della routine chiamata, che aggiusta gli stack ed esegue il ritorno al chiamante della subroutine originale. Questo è riportato in Figura 9.10. In questa figura è mostrato un solo livello, ma questo può essere ripetuto N volte. La Figura 9.11 riporta la codifica reale.

ESERCIZIO 4: (a) Se ci sono diversi livelli di subroutine chiamate, come appare lo stack SP?

(b) Che cosa fa SENTRY a C, Z, S, V, D e H? Come può essere usato?

(c) Scrivete SENTRY e SRPASS per il funzionamento segmentato. I nomi del registro indirizzo e la posizione dello stack possono essere definite simbolicamente in modo tale che la versione segmentata e non segmentata abbiano lo stesso codice sorgente eccetto per i valori assegnati nelle definizioni?

(d) Calcolate l'overhead in tempo (cicli) associato all'uso di SENTRY. Richiede una grande quantità di istruzioni per un programma relativamente semplice; può essere migliorata?

ESERCIZIO 5: Come dovrebbe essere cambiata TYWAIT se fosse usato lo schema di gestione dello stack, descritto prima, in congiunzione con le prestazioni del time-sharing precedentemente esaminate? Dove dovrebbe

essere memorizzato lo stack SP dell'utente? Ci sarebbe un qualche vantaggio nell'avere un'area comune per lo stack SP condivisa da tutti gli utenti? Se così fosse, che cosa occorrerebbe rimuovere dallo stack SP e salvare nell'area utente per mezzo di TYWAIT?

Un Meccanismo di Invio del SC

Abbiamo parlato ed usato l'istruzione SC nel Capitolo VIII, nella rou-

!Routine di entrata sistema

CALL SENTRY

Questa dovrebbe essere la prima istruzione di ciascuna routine: SENTRY si assicurerà che il valore di SR in uscita dalla subroutine sia lo stesso dell'entrata: esegue questa operazione assume il controllo e chiama il corpo della subroutine, come se fosse una subroutine di SENTRY.

SENTRY utilizza lo stack SP per ricordarsi del valore di SR da ricostituire: entra nel corpo della subroutine chiamante con l'indirizzo del suo codice uscita in cima allo stack SP e con il valore salvato di SR al successivo elemento, sotto. Un cambiamento di questo secondo elemento può permettere a SENTRY di ricostituire un valore di SR diverso da quello con cui era stato chiamato. Questo viene fatto con una chiamata a SRPASS che esegue in realtà una LD SP(#2),SR.

Nella versione segmentata il codice di SRPASS e SENTRY sarà quasi completamente diverso.

All'uscita, SENTRY esegue un TEST R0, per cui viene facilitato il passaggio dei risultati in R0. SENTRY non altera C, che può essere utilizzato per segnalare errori di stato.

```
!
SENTRY: EX SR,@SP                        !Salva SR, tiene in SR l'indirizzo
                                         di subroutine!
        PUSH @ SP,R0; LDA R0,SENX        !Mette l'indirizzo di SENX in SP!
        EX R0,@ SP
        PUSH @ SP,SR; LD SR,SP(#4)       !Mette l'indirizzo di subroutine, ri-
                                         costituisce SR!
        RET                              !RET falsa, per chiamare il corpo
                                         della subroutine!
SENX:   POP SR,@ SP; TEST R0; RET        !Ricostituisce SR pre-chiamato, e-
                                         segue il vero RET!

SRPASS: LD SP(#4),SR; RET
```

Figura 9.11 — Routine di Entrata del Sistema per la Gestione dello Stack

tine di inizializzazione TYIO, per simulare una interruzione da terminale. La SC ha due scopi principali:

- Fornire un mezzo ai programmi di modo «normale» per chiamare programmi del modo «sistema».
- Fornire una chiamata, ad una sola parola, per le routine usate frequentemente.

Quando viene eseguita l'istruzione SC, si verifica una trappola. Nel Capitolo II abbiamo discusso il meccanismo con cui vengono gestite le trappole: il PC, FCW e l'istruzione vengono poste nello stack e viene fatto un trasferimento ad un programma di elaborazione quando i valori di PC e FCW vengono memorizzati nell'area di stato del programma. (Vedasi Figure 2.8, 2.9).

L'istruzione SC può essere usata per un massimo di 256 diverse routine di sistema. Il byte meno significativo dell'istruzione contiene un valore compreso tra 0 e 255. Siccome l'istruzione si trova nello stack, la routine di elaborazione può usare quel byte come indice ad una tabella di indirizzi di routine di sistema. A questo punto si devono prendere alcune decisioni di tipo sistemistico:

- Le routine di sistema dovrebbero semplicemente essere trasferite o dovrebbero essere chiamate? Nel primo caso, ciascuna di esse deve eseguire una IRET; nel secondo caso chi gestisce l'istruzione SC esegue una IRET, per cui le routine di sistema sono delle subroutine che possono essere chiamate anche direttamente.
- Il gestore della SC dovrebbe salvare ogni registro (e ricostituirli se le routine di sistema sono delle subroutine?).
- Il gestore della SC dovrebbe predisporre ogni registro (esempio l'indirizzo di ritorno, per cui si potrebbero passare degli argomenti piazzandoli nella locazione di memoria che segue SC, o in modo tale che le routine possano ritornare, in funzione di alcune condizioni, diciamo, alla prima, seconda o terza locazione dopo SC?).
- Le routine SC come dovrebbero interagire con il meccanismo delle SENTRY/SRPASS?

Non possiamo rispondere senza un contesto siccome le risposte a queste domande possono differire in funzione delle diverse applicazioni. La Figura 9.12 riporta un gestore minimo di SC; tutto quello che fa è passare il controllo alla routine appropriata. Notate come il gestore trasferisca il controllo alla routine di sistema appropriata ponendo l'indirizzo della routine di sistema nello stack SP ed eseguendo una RET. Questo è un buon metodo per saltare da qualche parte del programma senza memorizzare l'indirizzo in un registro di una locazione di memoria «permanente». Notate anche che si utilizzano entrambi gli stack descritti prima, adottando

vantaggiosamente il nostro schema di gestione. Naturalmente si potrebbe fare la stessa cosa con un solo stack (utilizzando l'istruzione EX per ricostituire R1 e mettere nello stack l'indirizzo della routine di sistema, ma la codifica è un po' più lunga.

ESERCIZIO 6: (a) Modificate SCHAND in modo tale che utilizzi delle chiamate invece di trasferimenti: se volete arrivare alle routine di sistema con i registri inalterati dovrete usare un trucco simile a quello usato con SENTRY.

(b) Modificate SCHAND in modo che chiami le routine di sistema salvando R0 ed R1 nello stack SR e sostituisca i loro contenuti con i valori di FCW e PC posti nello stack SP. Quando la routine di sistema esegue il ritorno, SCHAND dovrebbe prendere i valori di PC e FCW, possibilmente modificati dalla routine di sistema, ed utilizzarli per aggiornare le coppie salvate nello stack SP dalla trappola SC. Poi dovrebbe ricostituire R0 ed R1 nello stack SR (dove essi possono essere stati modificati dalla routine del sistema) ed eseguire la IRET.

Quali sono i pro ed i contro di questo metodo?

(c) Scrivete una versione di SCHAND che salvi nello stack SR tutti i registri ad eccezione di HR, SR, SP, chiami la routine di sistema e li ricostituisca tutti prima di eseguire la IRET.

(d) Scrivete una versione di SCHAND con due tabelle: la tabella degli indirizzi delle routine di sistema ed una tabella di codici opzioni di un byte che identificano il modo con cui SCHAND dovrebbe interagire con le routine del sistema. I codici opzionali per una routine di sistema dovrebbero essere interpretati nel modo seguente:

Bit 0: Se è zero esegue il trasferimento (la routine esegue la IRET).
Se è uno esegue la chiamata (SCHAND esegue una IRET).

Bit 1: Se è uno salva i registri (eccetto HR, SR e SP) nello stack SR (e li ristabilisce prima di IRET, se c'è la opzione Call).
Se è zero non salva i registri.

Bit 2: Se è uno pone FCW e PC in R0 ed R1.
Se è zero non fa questo.

Bit 3: Se è uno aggiorna FCW e PC con i valori che vengono trasmessi in R0 ed R1 dalla routine di sistema.
Se è zero non fa questo.

Si intende che tutte queste opzioni sono specificabili in modo indipendente. Ci sono delle combinazioni contraddittorie?

(e) Tutti gli esempi sopra riportati son per il funzionamento non-segmentato. Come differiscono se vengono eseguiti in uno Z8002 o in uno Z8001 non-segmentato? (Vedasi Figura 2.7.) Come sarebbero in un segmento dello Z8001? Si possono localizzare tutte le differenze in SCHAND?

Le variazioni nell'esercizio precedente possono essere espanse all'infinito. Non c'è un modo ideale per usare queste prestazioni.

Nel Capitolo VIII abbiamo usato il comando

SC #IOGO

```
!Codice per elaborare la SC da trappola (versione non-segmentata)!

SCHAND: PUSH @SR,R1              !Salva R1 nello stack SR!
        LD R1,@ SP; CLRB RH1     !Indice della routine!
        PUSH @SP,SCTAB(R1)       !Mette l'indirizzo della routine nello
                                 stack SP!
        POP R1,@ SR; RET         !Ricostituisce R1 e ritorna!
```

Figura 9.12 — Gestore Minimo per SC

per realizzare la simulazione di una interruzzione di uscita per un terminale. Questa simulazione richiede una versione di SCHAND che non richiami la routine di sistema (o ha al limite una opzione per il trasferimento diretto); la versione della Figura 9.12 può essere usata per questa simulazione.

Era molto semplice capire il modo con cui questo sistema avrebbe lavorato in un ambiente time-sharing: IOGO è semplicemente il nome simbolico di un indice compreso tra 0 e 255; l'ingresso corrispondente nella SCTAB di Figura 9.12 sarebbe l'indirizzo TYOUT. Nella versione per il time-sharing TYOUT suppone che la «ragione» contenga nel suo byte più significativo il numero corrispondente al terminale. Per cui invece di andare direttamente a TYOUT, l'entrata corrispondente a IOGO dovrebbe portarci ad una routine TYSIM che ha una struttura di questo tipo:

```
!Metti il numero associato al terminale nella «ragione»!
        TYSIM:  EX R0,@SP
                LDB RH0,HR(#TRMNOF)
                EX R0,@SP
                JR TYOUT
```

Siccome le routine SC non sono routine dipendenti dal momento della interruzione come le TYOUT e TYIN che non possono assumere un valore corretto di HR, TYSIM può supporre che HR sia posizionata per l'utente corretto.

Inizializzazione del Sistema

Spesso l'«inizializzazione» è l'ultima parte del sistema che si scrive, questo si verifica perchè fino a quando non è stato scritto tutto il resto del programma non si sa che cosa debba fare esattamente l'inizializzazione.

Ciò che è stato presentato fino a qui non è un vero sistema di programmi, ma piuttosto un insieme di tecniche relative alla programmazione. Ma per un momento dimentichiamoci di questo, e guardiamo quale è il tipo di inizializzazione che richiede il nostro sistema quando viene acceso. La prima domanda a cui dobbiamo rispondere è: «Dove inizia l'esecuzione della CPU; quali sono i valori iniziali di PC e FCW da predisporre?».

La risposta è la seguente: la CPU preleva il suo stato iniziale dalla locazione 2 della memoria istruzioni, dove è memorizzata secondo il formato dello stack (vedasi Figura 2.7). Per uno Z8001 che utilizza una MMU per convertire gli indirizzi esiste un ulteriore complicazione: vedere se il prelievo iniziale dalla memoria dei valori di PC e FCW viene eseguito correttamente. La struttura circuitale della MMU si prende cura di questo. Si può disporre in modo che la MMU che converte normalmente «modo sistema, segmento zero» (o qualsiasi altra MMU per quanto importa) venga piazzato dal segnale di RESET nello stato in cui gli indirizzi vengono trasferiti alla memoria senza essere convertiti. Lo stato iniziale della CPU ci permette di (vedasi Figura 2.6) specificare:

- il funzionamento segmentato o non-segmentato
- il modo normale o sistema
- l'abilitazione o disabilitazione delle interruzioni vettoriali
- l'abilitazione o disabilitazione delle interruzioni non-vettoriali.

Siccome la nostra inizializzazione richiederà l'esecuzione di istruzioni privilegiate, dobbiamo predisporre il modo sistema. Dal momento che il puntatore dell'indirizzo dello stato del programma non è ancora fissato, dobbiamo specificare la disabilitazione delle interruzioni.

Il PC fissato da questo processo di RESET punta al nostro codice di partenza. La prima cosa che deve fare questo codice è di fissare il registro indirizzo dello stato del programma (vedasi Figura 2.9). L'area di stato del programma deve trovarsi in un blocco conosciuto di memoria *fisica* di programma, siccome non è ancora avvenuta l'inizializzazione della MMU. (Poiché vogliamo predisporre l'elaborazione della trappola di segmento prima di fare un qualsiasi uso della MMU, non vogliamo inizializzare la MMU prima di aver fissato il registro indirizzo dello stato del programma). I contenuti dell'area di stato del programma possono essere assemblati come parte del programma. Questo è un grande vantaggio che lo Z8000 ha rispetto a macchine del tipo PDP-11 che utilizzano locazioni fisse di memoria per l'elaborazione delle trappole. In tali macchine i vettori interruzione/trappola devono essere fissati per mezzo di codici espliciti, altrimenti potrebbero interferire con il funzionamento del caricatore. Nello Z8000, deve essere fissato esplicitamente solo il registro indirizzo dello stato del programma.

```
!Area di stato del programma
   Questa tabella definisce lo stato, della CPU, che deve essere fissato prima di
processare una qualsiasi delle interruzioni o trappole a cui può rispondere lo
Z8000. Questa è la versione non segmentata (utilizzabile solamente con lo
Z8000); per ciascuna interruzione o trappola c'è una parola di FCW e una parola
di PC.
!
SYSBIT = BIT14                  !Bit per sistema/normale!
SEGBIT = 0                      !Versione non-segmentata — non mette mai
                                 ad uno il bit di segmentazione!
SYSPRO = SYSBIT + SEGBIT        !Posizionamento per i programmi di «siste-
                                 ma»!

EVIBIT = BIT12                  !Abilita le interruzioni vettoriali!
ENVBIT = BIT11                  !Abilita le interruzioni non vettoriali!
ENABLE = EVIBIT + ENVBIT        !Abilita entrambi i generi!

PSA:  0;0                                !Non usate!
      SYSPRO + ENABLE; UNHAND            !Istruzione non implementata!
      SYSPRO + ENABLE; PIHAND            !Istruzione privilegiata!
      SYSPRO + ENABLE; SCHAND            !Chiamata sistema!
      SYSPRO; SEHAND                     !Errore di segmento — ferma le in-
                                          terruzioni!
      SYSPRO; NMHAND                     !NMI — sospende le altre!
      SYSPRO; NVHAND                     !NVI — sospende le altre!
      SYSPRO + ENVBIT                    !VI — permette il non vettoriale!
        TYIN                             !0: ingresso da tastiera!
        TYOUT                            !1: uscita per stampante/video!
        UNKNOWN                          !          •          !
            •                            !          •          !
            •                            !          •          !
            •                            !          •          !
        UNKNOWN                          !255: dispositivo sconosciuto!
```

Figura 9.13 — Area di Stato del Programma

Successivamente deve essere fissato il registro dello stack hardware (R15 o RR14); sia il registro di stack che il registro indirizzo dello stato del programma devono essere fissati prima che possa essere processata una qualsiasi trappola o interruzione. Il registro di stack deve essere fis-

sato ad un indirizzo conosciuto dello stack della memoria fisica, siccome la MMU sta ancora trasferendo degli indirizzi non convertiti.

Guardiamo come deve essere organizzata l'area di stato del programma. La Figura 9.13 mostra un'area di stato di un programma di un sistema con caratteristiche del tipo di quelle già discusse. Seguendo lo schema di Figura 2.9 (versione non-segmentata) viene organizzata una coppia di (FCW, PC) per ciascun tipo di interruzione o trappola. Le definizioni iniziali vengono usate per facilitare la predisposizione di FCW per le varie routine di processo. Non viene fatta alcuna predisposizione dei bit di FLAGS (esempio sono tutti posti a zero), ma essi potrebbero essere usati per trasmettere informazioni ad un gestore comune di trappola. Per esempio, se le istruzioni non implementate e le chiamate di sistema vengono gestite da una unica routine (invece delle routine separate UNHAND e SCHAND) allora, diciamo, che il bit C sarebbe posto ad uno nella FCW di una istruzione non implementata ed azzerato nella FCW della chiamata di sistema. Questo si potrebbe realizzare usando la definizione

CBIT = BIT7

e scrivendo

SYSPRO + ENABLE + CBIT

per porre ad uno la FCW dell'istruzione non implementata.

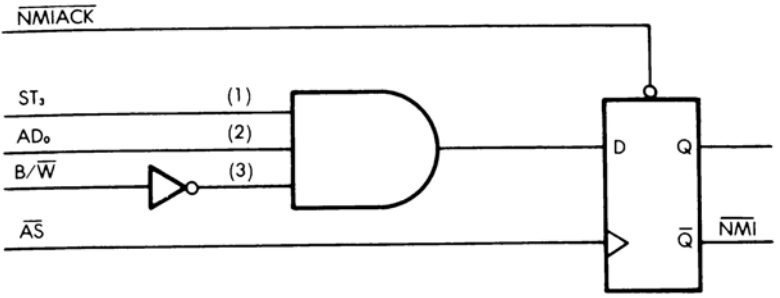
Delle routine di gestione specificate nel PSA di Figura 9.13, ci sono famigliari SCHAND, TYIN e TYOUT. Le altre dovrebbero essere spiegate. PIHAND, SEHAND e UNKNOWN sono le routine di gestione delle condizioni di errore: l'esecuzione di una istruzione privilegiata in modo normale, di un errore di segmento segnalato dalla MMU, di una interruzione vettoriale con un ID del dispositivo non assegnato.

La routine UNHAND può essere una routine di errore oppure può essere usata secondo lo schema del Capitolo IV — come una possibilità di chiamare fino ad ulteriori 6 insiemi di routine di sistema. Questo potrebbe eliminare alcune delle complicazioni delle diverse opzioni di progetto di SCHAND; potrebbero essere forniti diversi insiemi di routine con diverse tecniche di gestione.

Una delle possibili assegnazioni per le due interruzioni finali è:

NMHAND: trappola da «indirizzo di parola-dispari»

NVHAND: mancanza di alimentazione.



(1) ST_3 è uguale ad uno per accessi alla memoria, altrimenti è azzerato (vedasi Figura 2.12).
(2) AD_0 è posto ad uno per indirizzi dispari.
(3) $B/\overline{W}$ è azzerato per accessi a parole.

Siccome il PC non memorizza il bit zero, questo circuito non rivelerà i tentativi del programma di eseguire trasferimenti ad indirizzi dispari. Un tentativo di trasferimento ad un indirizzo dispari risulta in un trasferimento al successivo indirizzo pari più basso.

Nota Speciale: Nelle CPU prodotte prima di Gennaio 1980 lo stato di $B/\overline{W}$ può non essere valido durante la prima parola del prelievo dell'istruzione (ST_3-ST_0 con valore C_{16} – vedasi Figura 2.12). In questo caso, questo circuito può intrappolare alcuni prelievi di istruzioni che erano realmente validi.

Figura 9.14 — Una Implementazione Esterna della Trappola per «Indirizzi Dispari di Parola»

Perciò, NMHAND potrebbe riportare i tentativi di estrarre una parola o una parola-lunga di dato da un indirizzo dispari. Questo richiede la possibilità di implementare esternamente, come suggerito nel Capitolo IV, la prestazione dell'interruzione. La Figura 9.14 riporta la struttura schematizzata di questa prestazione.

Non entreremo in ulteriori dettagli a riguardo dei gestori delle interruzioni, siccome il loro comportamento dipende dai dettagli della loro specifica applicazione.

La Figura 9.15 mostra l'intera sequenza di inizializzazione che occorre per portarsi da una partenza a freddo ad un sistema, del tipo descritto prima in questo capitolo, funzionante in time-sharing. Le definizioni RAM = %4000 e RATE = 60*BIT9 rappresentano la predisposizione puramente arbitraria di parametri che variano da sistema a sistema. L'espressione 60*BIT9 definisce un valore di 60 in un campo che termina al bit 9 (il simbolo BIT9 viene scelto per definire dove si voglia il valore 512; esem-

pio 2^9). Predisponendo il registro di REFRESH al valore RE + RATE si abilita la generazione di cicli di rinfresco e si pone la frequenza a 60; esempio uno ogni 15 microsecondi con un clock di 4 MHz.

La linea $ABS 2 viene intesa come una istruzione all'assemblatore e al caricatore, per comunicare che questo codice deve iniziare alla locazione assoluta 2 della memoria. Lo pseudo-codice operativo realmente usato può variare da assemblatore ad assemblatore. Le due parole successive sono lo stato della CPU che devono essere predisposte dopo un RESET; il simbolo SYSPRO deriva dalla Figura 9.13.

Le tre chiamate alle subroutine INITMMU, INITERM e INITSS forniscono l'inizializzazione di tutte le prestazioni che abbiamo discusso. Non discuteremo INITMMU, ma daremo un'idea di cosa devono fare INITERM e INITSS.

INITERM fornisce l'inizializzazione dei terminali:

- Si deve organizzare la tabella, posta in CONTBL, degli indirizzi del blocco di contesto. (Figura 9.5)

```
!Sequenza delle istruzioni eseguite all'accensione a freddo (versione non segmentata)!

        RAM = %4000                 !Prima locazione della RAM!
        SYSTACK = RAM + 256         !Locazione iniziale dello stack!
        RE = BIT15                  !Abilità bit e RATE!
        RATE = 60*BIT9              !  per il registro di REFRESH!

        SABS 2
        SYSPRO; START
START:  LDA R0,PSA                  !Posiziona PSAP!
        LDCTL PSAP,R0
        LDA SP,SYSTACK; EI NVI      !Posiziona il registro di stack; abilita la caduta di tensione!
        LD R0,#RE + RATE            !Fa partire il registro di REFRESH!
        LDCTL REFRESH,R0
        CALR INITMMU                !Inizializza la MMU (solo se segmentato)!
        CALR INITERM; EI VI         !Posiziona i blocchi di contesto e gli anelli!
        CALR INITSS                 !Costruisce le aree dati utente!
        CALL TYWAIT                 !Inizia il timesharing — non torna più indietro!
```

Figura 9.15 — Codifica dell'Inizializzazione del Sistema

- Deve essere inizializzato ogni blocco di contesto: si devono determinare e predisporre gli indirizzi di I/O IOIN e IOUT e l'indirizzo del buffer di ingresso RING; alla FLAG dovrebbe essere dato il valore iniziale di DONE.
- Deve essere inizializzato ciascuno dei buffer ad anello; esempio fissate la lunghezza, azzerate il contenuto ed il puntatore posto ad un valore comune compreso tra 0 ed n – 1 (Figure 8.3, 8.5). (Per questo scopo dovrebbe essere aggiunta alla Figura 8.4 un'altra routine ad anello).
- Potrebbe essere utile inviare un carattere di inizializzazione a ciascuna stampante/video. In questo caso FLAG potrebbe essere posta ad OUTPUT e PTR potrebbe essere fissato con l'indirizzo corrispondente ad un NUL. Poi si potrebbe inviare un qualsiasi carattere (preferibilmente non-stampabile) a ciascun terminale, e quando ciascuno di questi è pronto ed interrotto, TYOUT predisporrà FLAG uguale a DONE (Figura 6.12).

Gli indirizzi di I/O, gli indirizzi dei buffer di ingresso e le loro lunghezze potrebbero essere assemblati in una tabella usata da INITERM come parte del programma, oppure la tabella potrebbe contenere solamente gli indirizzi di I/O e le lunghezze dei buffer, mentre le locazioni dei buffer potrebbero essere definite al momento dell'inizializzazione.

INITSS fornisce l'inizializzazione per il nostro sistema di time-sharing. I dettagli della routine dipendono dall'utilizzo dello schema a due stack descritto nel paragrafo della Gestione dello Stack. Siccome la nostra precedente descrizione del time-sharing era per il caso con un solo stack, descriveremo l'inizializzazione per la versione con un solo stack:

- Deve essere definita la tabella, posta a HRTB degli indirizzi dei dati dell'utente (Figura 9.7).
- Deve essere utilizzata ognuna delle aree dati utente (Figura 9.4): devono essere assegnati dei numeri utente compresi tra 0 e MXUSER – 1 (vedasi Figura 9.6) ed un numero di terminale; si devono determinare e fissare gli indirizzi dei buffer di linea di ingresso e uscita; si deve fissare il valore iniziale del registro di stack (usando HR(#SRF), non mostrato in Figura 9.4 – vedasi Figura 9.6).
- Deve essere inizializzata ogni linea di buffer; esempio fissata la loro lunghezza (Figura 8.8). (Per questo scopo si dovrebbe aggiungere un'altra routine di linea alla Figura 8.9).
- Si dovrebbe porre nello stack utente un indirizzo fittizio di ritorno («dummy»), come se la locazione prima dell'indirizzo di partenza contenesse una chiamata a TYWAIT.
- Siccome la chiamata di INITSS è seguita da una chiamata a TY-

WAIT, INITSS deve fissare un'area dati utente fittizia ed il valore di HR per TYWAIT con cui lavorare. Se INITSS fissa il numero MXUSER e se HRTB viene estesa di una parola per avere una posizione indicizzata da MXUSER, allora la sequenza di circolazione sarà:

MXUSER,1,2,..., MXUSER − 1,0,1,..., MXUSER − 1,0,1,...

In altre parole, INITSS al primo giro prende a prestito l'utente zero, che poi non utilizzerà più, per predisporre le cose.

Come per i buffer d'ingresso ad anello della tastiera, le aree dati utente e le linee di ingresso ed uscita dei buffer possono essere predeterminate al momento dell'assemblaggio del programma, o possono essere allocate al momento dell'inizializzazione.

ESERCIZIO 7: (a) Supponete che un clock sia collegato alla interruzione vettoriale di indice 2, in modo che si verifichi una interruzione ogni 100 ms. Scrivete una routine di gestione dell'interruzione per questo clock ed unitela al PSA mostrato in Figura 9.13. La vostra routine di gestione dovrebbe aggiornare l'orario della giornata memorizzato come ore, minuti, secondi e decimi. Dovrebbe aggiornare i decimi, e se necessario trasferire il riporto ai secondi, minuti e ore ed azzerare alla mezzanotte. A mezzanotte dovrebbe posizionare per la data un indicatore di «prossimo giorno» per l'utilizzo da parte di una routine di servizio separata.

(b) Usando ASK e le altre routine del Capitolo VIII scrivete una parte di codice per permettere che l'utente posizioni il tempo del giorno.

ESERCIZIO 8: (a) Supponete di voler eseguire tutti i nostri programmi in modo normale eccetto le routine di gestione interruzione e trappola e tutte le routine chiamate da SCHAND. Come dovrebbe cambiare la nostra inizializzazione? Avremmo bisogno di usare il registro di controllo NSP? Potremmo evitare un'area utente fittizia per la routine INITSS usando una istruzione LDPS per portarci nel mezzo di TYWAIT e contemporaneamente portarci in modo normale?

(b) Scrivete INITERM e INITSS supponendo che le tabelle assemblate definiscano le allocazioni della memoria.

(c) Escogitate una allocazione di memoria per la struttura a due stack e variate coerentemente TYWAIT e INITSS. Che cosa c'è ancora da cambiare?

(d) Fornite le versioni segmentate delle routine di questo capitolo.

Capitolo X

L'Ambiente di Sviluppo dei Programmi

Nei precedenti capitoli di questo libro abbiamo discusso gli elementi dello Z8000 che devono conoscere i programmatori. Abbiamo illustrato il processo che porta dalla scrittura di un algoritmo alla scrittura del programma; abbiamo visto come siano i programmi dello Z8000 per le varie applicazioni; ed abbiamo usato e discusso gli importanti principi di progettazione della moderna ingegnerizzazione del software. Tutte le cose dette sono comuni alla programmazione di tutti gli Z8000. In questo capitolo discuteremo la parte variabile: il particolare insieme di strumenti hardware e software con cui deve lavorare ciascun programmatore per sviluppare dei programmi — l'ambiente di sviluppo dei programmi.

Gli strumenti essenziali di sviluppo sono: un editore di testo (editor), un assemblatore/caricatore e un debug (correttore di errori). Un editor, o come è stato chiamato recentemente word processor (elaborazione della parola), è un programma che consente di esaminare ed aggiornare un blocco di testo tenuto dalla macchina in forma leggibile. L'analisi e l'aggiornamento vengono qualche volta chiamati «manutenzioni» ed un blocco di testo tenuto in forma leggibile dalla macchina è detto text files (flusso di informazioni o dati che compongono un testo): per cui l'editore di un testo è un programma che mantiene i file di testo. Anche la stampa dei file di testo è considerata qualche volta una funzione dell'editor, ma in realtà è una funzione diversa. Un «vero» programma di stampa dei file di testo, nei vari formati, può servire molti moduli del sistema di sviluppo — non solo l'editor.

L'uso principale del text editor, in un ambiente per lo sviluppo dei pro-

grammi, è preparare e mantenere i file *sorgente* (source file). Questi sono i file contenenti i programmi. Per esempio, tutti i testi di Figure 6.11, 6.12, 6.13, 6.14 e 8.2 potrebbero costruire tutti insieme un file sorgente per un programma di I/O per un terminale.

L'assemblatore è il programma che traduce i file sorgente in un vero codice macchina, perciò, esso specifica i valori fissati per ogni bit di ogni byte della porzione di memoria destinata al programma che viene assemblato. L'assemblatore e l'editore devono lavorare con un comune formato del file sorgente. In altre parole l'editore deve preparare i file con lo stesso formato con cui l'assemblatore si aspetta di riceverli tramite il file sorgente. D'altra parte il codice macchina generato dall'assemblatore, chiamato *object file* (file oggetto), deve essere in un formato riconoscibile dal caricatore. Il caricatore è il programma che converte l'uscita dell'assemblatore nel programma reale che risiede nella memoria fisica del calcolatore. Questo può essere un semplice processo di caricamento dell'uscita dell'assemblatore, byte per byte, nella memoria o può essere il caricamento dell'insieme dei file di uscita dell'assemblatore e dei file ottenuti legando gli stessi in modo prestabilito.

Le funzioni dell'assemblatore e del caricatore non sono distinte. Esse costituiscono due parti di uno stesso processo: la conversione del codice sorgente scritto dal programmatore nel corrispondente insieme di istruzioni espresse in linguaggio macchina, esprimono due esempi distinti di modularità:

- L'uscita dell'assemblatore (file oggetto) può essere preparata in un certo istante (e con una certa configurazione di hardware) e caricata in un secondo tempo (e su di una configurazione diversa, possibilmente più piccola o più grande).
- Il codice macchina può essere ottenuto da vari file oggetto permettendo in tal modo che le parti logicamente separate di un grosso programma siano editate ed assemblate separatamente, permettendo inoltre ai programmatori di ampliare i loro programmi con selezione di subroutine di utilità ricavate dalle «librerie programmi».

La prestazione di debug, come suggerisce il nome, è uno strumento per eliminare gli errori dal programma. Nella sua forma più semplice è costituito da un programma che ricerca i contenuti dei registri selezionati e delle locazioni della memoria dati nei vari punti di funzionamento del programma in «debug». Nella sua forma più sofisticata è un sistema hardware/software che analizza gli stati della CPU e molti dei suoi bus e dispositivi periferici ad esso associati. Esso salva questi stati per un arco di tempo posto prima e dopo gli eventi specificati come di «sincronizzazione» (trigger).

La prestazione debug può essere completamente indipendente dall'assemblatore/caricatore e dalla sua uscita. In questo caso tutti gli indirizzi e i contenuti vengono considerati come numeri binari o esadecimali e tutta l'interpretazione è lasciata al programmatore. Spesso questo è tutto ciò con cui deve lavorare il programmatore, ma un programma migliore di debug sarà in grado di assemblare o disassemblare le istruzioni usando i mnemonici e la sintassi dell'assemblatore e anche i simboli specifici definiti nei file sorgente per il programma esaminato.

Nei paragrafi seguenti discuteremo in maggior dettaglio le caratteristiche e le opzioni di questi strumenti fondamentali allo sviluppo di un programma.

Editori di Testo

In generale gli editori di testo hanno ben poco a che fare con il calcolatore per cui sono stati generati i file sorgente. Invece, gli editori più popolari possono essere utilizzati su diversi calcolatori. Per esempio, l'editore TECO sviluppato in un primo tempo per il DEC PDP-10, è ora disponibile per l'utilizzo con molti altri calcolatori. Ci sono però alcuni dubbi sul fatto che presto ci sarà un TECO disponibile per l'uso su sistemi Z8000. Anche i popolari sistemi «word-processing», utilizzati da molto personale impiegatizio dovrebbero presto divenire di largo uso nella preparazione di file sorgente. Nel frattempo gli editori che più probabilmente saranno disponibili per essere utilizzati con lo Z8000 saranno adattamenti di quelli precedentemente sviluppati o adattati per l'uso con lo Z80, 8080, e altri popolari microprocessori. Questo è vero specialmente per gli editori di sistemi di sviluppo Z8000 che funzioneranno con microcalcolatori basati sullo Z80.

Le operazioni fondamentali di un editore sono:

- La creazione di un nuovo testo
- La cancellazione di un vecchio testo
- Il riarrangiamento di testi esistenti
- La sistemazione o la sostituzione ad hoc di una parte di un testo con un altro
- La gestione di vari file e dei mezzi di memorizzazione che contengono il testo su cui si sta lavorando.

Ci sono tre caratteristiche principali che differenziano i vari editori:

- Se considerano il testo con cui lavorano come stringhe di testo o come insieme di linee
- La potenza e la flessibilità del loro meccanismo di specificazione del comando
- Se permettono di saltare da un posto all'altro del file di testo a piaci-

mento o se essi richiedono di esaminare il file dall'inizio alla fine; generalmente una «pagina» alla volta.

In generale si può parlare molto degli editori, ma quanto è stato detto finora è di limitato uso diretto ai programmatori dello Z8000. Una caratteristica comune a tutti gli editori è che i loro comandi sembrano a prima vista difficili, ma diventano presto semplici all'utilizzatore assiduo.

Assemblatori

Diversamente dall'editore, un assemblatore è strettamente legato ai dettagli del particolare calcolatore per cui viene fornito. Suo scopo è di produrre un codice macchina da file sorgenti costituiti da istruzioni fornite con mnemonici specificati dal costruttore e simboli definiti dal programmatore per le costanti e gli indirizzi di memoria. Al limite ogni costruttore di calcolatori fornisce un assemblatore per ognuno dei suoi calcolatori; inoltre sono generalmente disponibili degli assemblatori addizioniali da parte di venditori esterni. Le variazioni sul tema di base definito sopra sono le seguenti:

- Macro
- Assemblaggio condizionato
- Simboli locali
- Controllo di strutture
- Codice rilocabile
- Supporto per stringhe di testo
- Operazioni aritmetiche e logiche su simboli

Attraverso questo libro abbiamo presentato programmi che assumono implicitamente l'esistenza di un assemblatore; delle caratteristiche elencate prima, solamente le ultime due sono state utilizzate in ogni nostro programma.
Per esempio, in Figura 6.13

ESCS: BYTES('$', '/',TRM)

presuppone una minima gestione della stringa di testo. La maggior parte degli assemblatori permetterà qualcosa del tipo

ESCS: ASCII «$/»

con l'inclusione opzionale di un carattere per la terminazione della stringa (sopra TRM) specificato in un qualche modo.

Facciamo uso delle operazioni aritmetiche su simboli con espressioni, ricavate dalla Figura 9.15, del tipo

SYSTACK = RAM + 256

RATE = 60*BIT9

molti assemblatori supportano tali espressioni elementari, ma in alcuni altri non si possono usare le espressioni che utilizzano dei simboli per i registri (esempio vedasi Figura 6.11).

Nel Capitolo III abbiamo accennato ai problemi che rendono desiderabili i simboli locali. Con la maggior parte degli assemblatori, ad ogni simbolo si può far riferimento da un qualsiasi altro punto dello stesso file sorgente, ma alcuni assemblatori forniscono un mezzo per restringere l'uso di un dato simbolo ad una porzione specifica del file sorgente, esempio alla subroutine in cui è definito.

La modularità è legata alla possibilità di avere dei simboli locali. Un altro aspetto della modularità ha a che fare con il codice rilocabile. L'assemblatore tiene cura di assemblare a specifiche locazioni di memoria le istruzioni che appaiono nel file sorgente; il programmatore fa riferimento ad ogni locazione desiderata indirizzandola con nomi simbolici. L'assemblatore assegna dei valori a questi nomi (vedasi Figura 3.1) e li assembla assegnando i valori opportuni alle istruzioni che li usano. Tutto questo è legato all'assemblaggio di tutto il programma in un'unica passata.

Se il programmatore desidera combinare dei programmi o di suddividere un programma in parti logicamente indipendenti, allora non può più assegnare le locazioni di memoria o assemblare i valori reali riferendosi ai simboli in una parte di programma con le istruzioni in un'altra parte. In questo caso, l'assemblatore produce un codice *rilocabile*, ed il *caricatore rilocabile* (chiamato anche linkage editor) deve assemblare i valori reali delle istruzioni di una parte che si riferiscono ad un'altra parte. Questo programma prende il file oggetto in uscita dall'assemblatore ed assegna a questi le locazioni reali di memoria (esempio li riloca). È compito del programmatore *dichiarare* quali simboli di un dato file sorgente vengono richiamati da un altro file sorgente e quali simboli richiamati in un dato file sorgente si aspetti vengano forniti da altri file sorgente. I simboli che vengono richiamati da altri file sorgente sono chiamati *global* o *entry*; i simboli che devono essere forniti da altri file sorgente sono chiamati *external*. Perciò è compito del programmatore dichiarare le global, entry o external.

Nel Capitolo III abbiamo parlato di come implementare istruzioni del tipo

IF (così e così) THEN (una cosa) ELSE (l'altra)

per mezzo di test e salti. (Vedasi Figura 3.2.) Istruzioni di questo tipo sono generalmente disponibili in linguaggi ad alto livello tipo PASCAL o C, ma si trovano raramente nei linguaggi assemblatori.

I rimanenti due elementi della nostra lista sono i più importanti: macro ed assemblaggio condizionato. Per comprenderli dovremo guardare l'assemblatore sotto un'altra ottica. Pensate all'assemblatore come ad una macchina alla quale il file sorgente sta fornendo un carattere alla volta. Questa successione di caratteri è chiamata *input stream* (flusso d'ingresso) dell'assemblatore. La macro e l'assemblaggio condizionato sono modi per alterare il flusso di ingresso dell'assemblatore.

Una istruzione di assemblaggio condizionato prende la forma

IF (condizione) THEN *testo*

Se la condizione è vera, allora il carattere del *testo* è il prossimo pezzo del flusso di ingresso; altrimenti, il flusso continua con l'istruzione successiva all'istruzione di assemblaggio condizionato. Analogamente sono gestite varianti del tipo

IF (condizione) THEN *testo 1* ELSE *testo 2*

Una macro funziona come una sostituzione di testo. Quando nel file sorgente appare una particolare sequenza di caratteri, nel flusso di ingresso viene posta una predeterminata stringa di sostituzione. Questo può essere ripetuto ad ogni livello, perciò, se la stringa sostituita contiene delle macro, allora verranno utilizzate le stringhe che sostituiscono queste macro. Le stringhe di sostituzione predeterminate sono specificate nelle definizioni delle macro. Diamo qui di seguito un esempio.

Nella Figura 6.13 si noti che appare due volte la sequenza di istruzioni

LD R1,LINE; LDB RL0,CHAR; CALL ADLINE

in ECS ed in ECC. Supponiamo, che all'inizio della Figura 6.13 o all'inizio della Figura 6.12 o anche in Figura 6.11 sia stata messa la definizione

PUTLINE = «LD R1,LINE; LDB RL0,CHAR; CALL ADLINE»

Con questa definizione di macro potremmo scrivere PUTLINE in ciascuno dei due posti dove appariva la precedente sequenza di tre istruzioni. Il file sorgente sarebbe quindi più breve e più facile da leggere. Le istruzioni reali generate non cambierebbero, siccome ogniqualvolta che si incontra PUTLINE verrebbero immesse nel flusso di ingresso le tre istruzioni originali.

Adesso guardiamo ancora alla Figura 6.13. In ECCR appare colà una sequenza

```
LD R1,LINE; LDB RL0,#TRM; CALL ADLINE
```

che a parte #TRM invece di CHAR è identica alla sequenza precedente. Supponiamo ora che la nostra definizione si legga come

```
PUTLINE x = «LD R1,LINE; LDB RL0,x; CALL ADLINE»
```

Con questa definizione potremmo scrivere

```
PUTLINE CHAR
```

per le prime due macro e

```
PUTLINE #TRM
```

per la più recente. In questo caso si è chiamato x il *parametro* della macro PUTLINE.

Per ultimo riportiamo un esempio di una combinazione di assemblaggio condizionato e di macro

```
SUM x,y =   IF x è un registro THEN
              «ADD x,y»
            ELSE
              «PUSH @ SR,R0
              LD R0,x; ADD R0,y; LD x,R0
              POP R0,@SR»
```

Per cui SUM R0,R1 genera ADD R0,R1; ma se TEMP è una locazione di memoria, allora SUM TEMP,R1 genera

```
PUSH @ SR,R0
LD R0,TEMP; ADD R0,R1; LD TEMP,R0
POP R0,@SR
```

Questo ultimo esempio mostra solo la punta dell'iceberg. La combinaziaone di una capacità di macro con l'assemblaggio condizionato consente al programmatore di trasformare il linguaggio di assemblaggio originale in un potente strumento adatto alle specifiche applicazioni sottomano.

La sintassi qui presentata per l'assemblaggio condizionato e per la definizione delle macro non è quella di uno specifico assemblatore e non esiste una uniformità o standardizzazione in questo settore. Dovrete ogni volta imparare la sintassi adatta al vostro particolare sistema.

Nel Capitolo I abbiamo detto che le subroutine sono il primo strumento importante del programmatore. Le macro, specialmente con l'assemblaggio condizionato, sono il secondo.

Strumenti per il debug

Il *dump* ed il *trace* erano i primi strumenti dei debug utilizzati in ambiente di elaborazione batch e sono ancora in uso oggigiorno. Il dump, nella sua forma più cruda, è un elenco esadecimale dei contenuti di tutte le locazioni di memoria che hanno una qualche relazione con il programma eseguito. Il trace è un meccanismo mediante il quale viene modificato il programma in esecuzione (o eseguito in modo interpretativo da un programma di simulazione) tale che l'informazione scelta venga stampata a punti prefissati, durante l'esecuzione del programma. Per esempio, i contenuti di certe locazioni di memoria potrebbero essere stampati dopo l'esecuzione di una passata di un loop, o si potrebbero stampare i nomi (o gli indirizzi esadecimali) di determinate subroutine ogni volta che vengono chiamate.

Quando sono stati sviluppati gli ambienti di programmazione interattiva, è diventato possibile usare i debug interattivi. Il primo di questi, largamente usato, è stato il ODT per il PDP-8. ODT sta per Octal Debug Tape (Nastro di Debug Ottale): ottale perchè gli indirizzi a 12 bit e i contenuti delle parole del PDP-8 sono sempre guardati come quattro cifre ottali invece che tre cifre esadecimali; nastro, perchè questo programma era fornito su nastro perforato, per potere essere caricato usando il lettore di nastro perforato della consolle della telescrivente. Le caratteristiche principali di un debug interattivo sono:

- Controllo e modifica di locazioni di memoria prefissate
- Punti di interruzione del programma (breakpoints)

La prima di queste caratteristiche, rappresenta una evoluzione del dump; la seconda è l'evoluzione del trace. Diamo qui di seguito un esempio di come potreste utilizzare un debug interattivo con un programma dello Z8000.

Supponete di controllare il funzionamento della routine SAY di Figura

8.10. Desiderate conoscere se essa struttura nel modo opportuno la tabella per la chiamata di TYIO. Supponiamo che il vostro debug sia caricato in memoria all'indirizzo 2000, il programma di Figura 8.12 all'indirizzo 1000 e le routine di Figura 8.10 caricate in modo che l'istruzione CALL TYIO di SAY si trovi all'indirizzo E16. Iniziate partendo dall'indirizzo 2000. Il debug si posizionerà per ricevere un comando e voi lo istruirete per porre un breakpoint in E16, perciò, desiderate far eseguire il programma fino a quando viene eseguita TYIO. Poi volete controllare la tabella che ha posizionato SAY nello stack.

Il debug, dopo avere notato la vostra richiesta, si predispone per un altro comando, e voi gli direte di iniziare ad eseguire il programma posto all'indirizzo 1000. Quasi immediatamente sarete avvertiti che l'esecuzione ha raggiunto il breakpoint in E16. Esaminate R1 per sapere dove è memorizzata la tabella TYIO; poi esaminate le locazioni di memoria che partono da quell'indirizzo per vedere se sono come sperate. Se è così, potete gestire il debug per piazzare un altro breakpoint da qualche parte più avanti nel proceso di esecuzione e continuare l'esecuzione dal punto dove era posto il breakpoint originale (esempio E16). Se non volete, potete voler piazzare il breakpoint in qualche punto del processo antecedente al vecchio breakpoint e dire al debug di ripartire da 1000.

Non abbiamo discusso come il debug realizzi i breakpoint. Un modo ovvio consisterebbe nel dedicare una delle 256 entrate di SC a questa funzione. Poi il debug, appena prima di trasferire il controllo al programma in esame, potrebbe sostituire i contenuti delle locazioni di breakpoint specificate con l'appropriata istruzione SC. Questo richiede che si possa scrivere nella zona di memoria dedicata a quella istruzione e che la memoria dell'istruzione del programma si trovi nella memoria dati nel debug. Il modo più facile per assicurare quest'ultima cosa è di evitare di avere gli spazi dati ed indirizzi delle istruzioni separati.

Il debug interattivo è utile in quei casi in cui è ben definito il flusso di controllo della situazione in esame e dove l'azione del debug non altera il comportamento della CPU o del programma in modo tale da alterare il testo in uscita. Quando le interruzioni o il time-sharing influenzano la situazione che si sta esaminando o quando si verificano sintomi tipo i cambiamenti inspiegabili delle locazioni di memoria, si deve ricorrere ad un altro genere di strumento. Questo è chiamato in-circuit emulator (ICE — emulazione del circuito) che è una derivazione della «scatola trappola» usata per i minicalcolatori.

L'ICE viene posto direttamente nello zoccolo del dispositivo di CPU (al posto della CPU) ed invia indietro dei segnali ad un sistema di registrazione e monitoraggio che si «sincronizza» su certe combinazioni di segnali

presenti sulle linee della CPU e su altre linee prefissate di segnali. Gli stati di questi segnali vengono salvati a ciascun ciclo di clock, ed esaminati in un secondo tempo in una specifica «finestra» attorno all'evento di sincronizzazione.

In origine, come per i loro predecessori (le scatole trappole), gli ICE erano progettati per uno specifico microprocessore. La maggior parte dei moderni emulatori usa una logica generale di sincronizzazione e registrazione che con «piastre di personalizzazione» determinano quale sia il microprocessore emulato. Quindi, l'interazione del programmatore con l'emulatore sarà specifica al sistema di emulazione, e non al microprocessore emulato.

Hardware per lo sviluppo di programmi

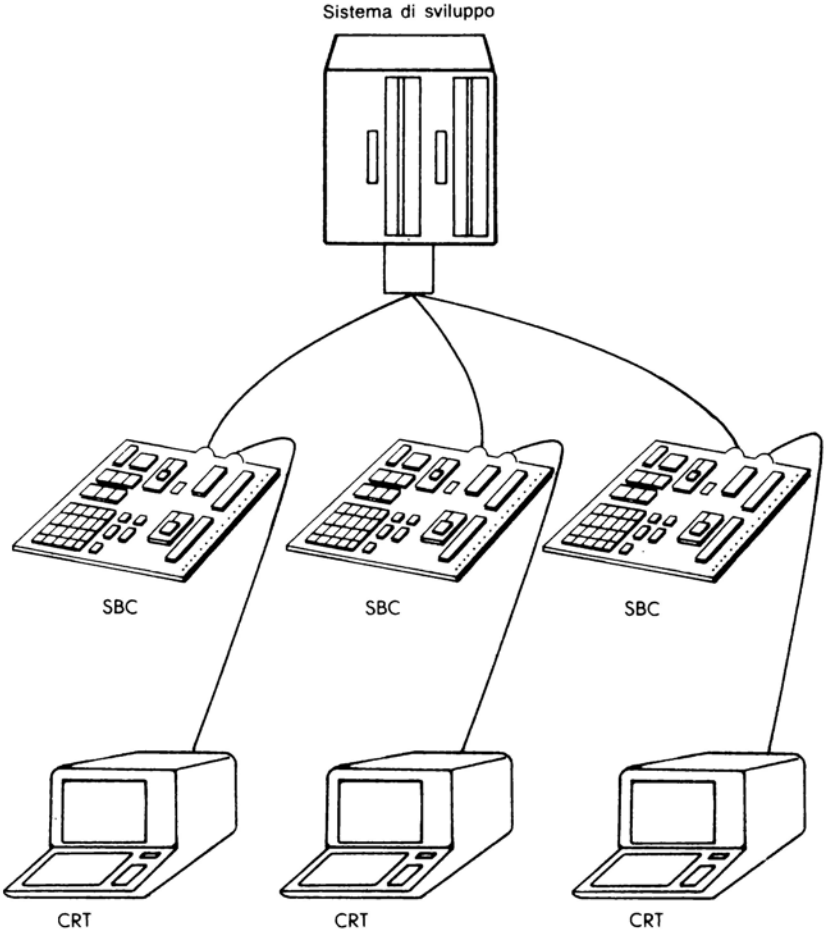
Ci sono due tipi di hardware utilizzati per lo sviluppo di programmi per microprocessore: i sistemi di sviluppo e i «single-board computer» (calcolatori di scheda singola). Il sistema di sviluppo è una configurazione completa con unità di memorizzazione secondarie (esempio floppy-disk) e molte RAM per far girare gli editori e assemblatori. Un calcolatore single-board ha una CPU, alcuni circuiti di I/O, una piccola quantità di RAM, una ROM contenente un debug esadecimale, un caricatore ed un'area wire-wrap per lo sviluppo dell'hardware prototipale.

I programmi che devono essere eseguiti su di un calcolatore su singola scheda devono essere caricati da un terminale usando il debug esadecimale o caricati dal lettore di nastro perforato o dall'unità a cassetta del terminale nel formato codice macchina, dopo averlo assemblato da qualche altra parte. Una variante a questa ultima alternativa è un caricamento da linea di trasmissione (down-line load) in cui il nastro perforato, o la cassetta, è simulato dal sistema di sviluppo. Se si devono eseguire solo programmi molto piccoli sul calcolatore single-board, questi possono essere assemblati manualmente usando il metodo illustrato nel Capitolo IV; in questo caso si può fare a meno del sistema di sviluppo. Ma questo ha senso solamente se i programmi dello Z8000, considerati, sono molto piccoli e cambiati raramente.

D'altra parte, l'utente del sistema di sviluppo dello Z8000 non può fare a meno del calcolatore single-board ed eseguire programmi sul sistema di sviluppo dello Z8000 per una ragione molto semplice: il sistema di sviluppo può non avere dentro uno Z8000. Per esempio, il primo sistema di sviluppo per lo Z8000, commercialmente disponibile, è basato sullo Z80. Questa non è una situazione atipica: vera anche per sistemi di sviluppo di altri microprocessori. Dopo tutto, non fa molta differenza il tipo di CPU

che esegue l'editor o l'assembler dello Z8000. Come per lo ICE, è molto più facile trapiantare una nuova CPU in un sistema stabilizzato piuttosto che costruire un nuovo sistema per essa.

La Figura 10.1 mostra una possibile configurazione di sviluppo. Ciascun calcolatore single-board ha il proprio terminale, e ciascuno di essi



Diversi calcolatori single-board connessi uno alla volta ad un sistema di sviluppo. Quando SBC è collegato il terminale del SBC diviene il terminale del sistema di sviluppo.

Figura 10.1 — Una Possibile Configurazione per lo Sviluppo dei Programmi

può essere collegato, uno alla volta, al sistema di sviluppo per editare, assemblare e ricaricare il programma da linea. Se il programmatore desidera farlo, ci sono dei calcolatori Z8000 single-board che consentono ai terminali ad essi collegati di collegarsi direttamente con il sistema di sviluppo.

Casualmente notiamo che i primi calcolatori single-board, disponibili commercialmente, usano i circuiti SIO, PIO e CTC della famiglia Z80. Si può prevedere che le versioni successive di questi calcolatori single-board potranno usare i circuiti della famiglia Z8000.

APPENDICE A

Risposta agli esercizi selezionati

Capitolo I

1. Il passo 3 viene eseguito quattro volte. I passi 1, 2, e 7 sono eseguiti una volta ciascuno. Il passo 3 sarebbe eseguito sei volte se 112 fosse in cima e 12573 fosse in fondo.

3. $7 = 111_2$. $196_{10} = 11000100_2$.

4. $196_{10} = 11000100_2 = 1100_2 \times 2^4 + 0100 = C \times 16 + 4 = C4_{16}$.

5. Tre bit permettono di distinguere Otto elementi ($8 = 2^3$). Per rappresentare dodici stati occorrono quattro bit ($2^4 = 16 > 12$, $2^3 = 8 < 12$). Quattro cifre esadecimali possono rappresentare $16^4 = 65536$ numeri differenti. Otto cifre esadecimali possono rappresentare $16^8 = 4.294.967.296$ numeri differenti.

6. $196 \div 16 = 12$ con resto 4. La cifra finale è $4.12 \div 16 = 0$ con resto 12. La prima cifra è C (=12) perciò, $196_{10} = C4_{16}$.

7. B + B = 16. 1A + B1 = CB. $A^2 = 64$. 1B93F ÷ 2B5 = A3.

8. − 5 è rappresentato da FFFB. − 7FFF = 8001. − 1A1 = FE5F. Perciò, FF7 − 1A1 = FF7 + FE5F = E56. (− 1) + (− 5) = − 6. (− 7FFE) + (− 6) = + 7FFC.

9. 6 − 4 = 2; C = 0. BB − BC = − 1; C = 1. FFFF + 3 = 2; C = 1. FFFF + FFFF = FFFE; C = 1.

10. V = 0 per 6 − 4, BB − BC, FFFF + FFFF. (− 17) + (− 7FF1) = + 7FF8; C = 1; V = 1.

11. (a) Inizializza N a zero.
(b) Riceve la cifra successiva. Se «x», allora fai.
(c) Moltiplica N per A (= 10) e somma il valore alla cifra. Sostituisce N con il risultato di questo calcolo.
(d) Ritorna al passo (b).

12. Sedici bit possono rappresentare 4 cifre BCD. Perciò, 16 bit possono rappresentare 10000 cifre BCD ($10000 = 10^4$). Sedici bit possono rappresentare 65.536 numeri esadecimali ($65.536 = 16^4$). La regola è: 4n bit possono rappresentare 10^n numeri BCD o 16^n numeri esadecimali. Il rapporto è $(10/16)^n$.

15. «Hi! I'm a Z8000.» può essere rappresentato in ASCII con: 48, 69, 21, 20, 49, 27, 6D, 20, 61, 20, 5A, 38, 30, 30, 30, 2E. (Vedasi Figura 1.15).

Capitolo III

1. 48 ∨ 0 = 48. 48 ∨ 48 = 0. F0 ∨ 0F = FF.

2. 4 mod 3 = 1.

3. La versione crittografata di «Hi! I'm a Z8000.» è: 29, 0B, 53, 41, 2A, 46, 09, 41, 03, 52, 3B, 59, 52, 42, 51, 4D, 0. Tre di questi caratteri non sono stampabili: 0B (VT), 09 (HT), 03 (ETX). Sostituite ciascuno di questi con «.» ottenendo il testo seguente:

).SA*F.A.R; YRBQM

4. $-x - 1 = -(x + 1)$.

Capitolo VI

1. (a) ADDL RRO, #1 richiede 14 cicli.
JR NZ,x richiede 6 cicli.
Perciò, ciascun giro del loop usa 20 cicli, esempio 5μ s a 4MHz.
$(2.4 \times 10^8) \times (5 \times 10^6) = 1.2 \times 10^3$ secondi = 1200 secondi = 20 minuti.

(b) 10 cicli a 4 MHz = 2.5 μ s.

3. (a) 100 ms/secondo ÷ (11 bits/carattere × 10 carattere/secondo) = 1000/110 ms/bit = 9.091 ms/bit.

Capitolo VII

2. Sottrarre 400 equivale ad aggiornare il clock di 10 ms. La quantità rimanente dopo la sottrazione è una parte legittima dei prossimi 10 ms. Se

invece di sottrarre 400 si azzerasse, si determinerebbe una perdita di tempo. Sarebbe perso tutto il tempo trascorso tra il momento in cui il contatore raggiunge 400 ed il momento in cui viene azzerato.

Capitolo VIII

1. Se TYENT si trova nel suo loop a TYWSM, allora TYWAIT non finirà mai il suo test, essendo in atto il completamento di un'interruzione, viene sospesa l'esecuzione di TYWAIT.

5. SUFTAB: 1, «st»
2, «nd»
3, «rd»
21, «st»
22, «nd»
23, «rd»
31, «st»
0, «th» ! Default = «th», e.g., 4th, 5th, etc.!

APPENDICE B

Tabella dei caratteri ASCII

COD.	CARATT.	COD.	CARATT.	COD.	CARATT.	COD.	CARATT.
00	NUL	20[1]		40	@	60[5]	
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27[2]	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	TAB	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C[3]	,	4C	L	6C	l
0D	CR	2D	—	4D	M	6D	m
0E	SO	2E		4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A		5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D[6]	}
1E	RS	3E	>	5E	↑	7E	~
1F	US	3F	?	5F[4]	←	7F[7]	RUBOUT

[1] spazio
[2] apice
[3] virgola
[4] o sottolineatura
[5] accento
[6] o ALT MODE
[7] o DEL

APPENDICE C

Codifica dei Campi delle Istruzioni

Campo di modo	Registro Src	Modo di Indirizzamento	Registro Dst	Modo di Indirizzamento
0	1-15	Indiretto su Registro ( )	0-15	Registro Indiretto (@)
0	0	Immediato (#)		
4	1-15	Indicizzato (X)	1-15	Indicizzato (X)
4	0	Indirizzo diretto (DA)	0	Indirizzo diretto (DA)
8	0-15	Registro (R)	0-15	Registro (R)

Codifica del Modo di Indirizzamento Mediante Modo e Registro

Codice Condizione (esadecimale)	Significato	Mnemonico	Codice Condizione (esadecimale)	Significato	Mnemonico
0	Falso	Nessuno	8	Vero	Spazio*
1	S XOR V = 1	LT	9	S XOR V = 0	GE
2	LT OR Z = 1	LE	A	LT OR Z = 0	GT
3	C OR Z = 1	ULE	B	C OR Z = 0	UGT
4	V/P = 1	OV,PE	C	V/P = 0	NOV,PO
5	S = 1	MI	D	S = 0	PL
6	Z = 1	Z,EQ	E	Z = 0	NZ,NE
7	C = 1	C,ULT	F	C = 0	NC,UGE

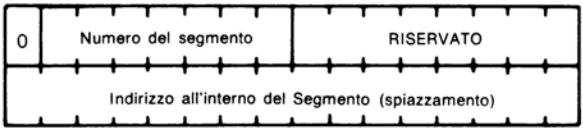
* Caso di Default (esempio JR X ha 8 nel campo cc)

Codici Condizione

ESADEC.	SIGNIFICATI	ESADEC.	SIGNIFICATI
0	R0,RH0,RR0,RQ0	8	R8,RL0,RR8,RQ8
1	R1,RH1	9	R9,RL1
2	R2,RH2,RR2	A	R10,RL2,RR10
3	R3,RH3	B	R11,RL3
4	R4,RH4,RR4,RQ4	C	R12,RL4,RR12,RQ12
5	R5,RH5	D	R13,RL5
6	R6,RH6,RR6	E	R14,RL6,RR14
7	R7,RH7	F	R15,RL7

Interpretazione dei Campi di Registro delle Istruzioni

INDIRIZZO NON-SEGMENTATO
16 bit indirizzano 65.536 byte



INDIRIZZO SEGMENTATO ESTESO
23 bit indirizzano 8.388.608 byte
il numero del segmento a 7 bit indirizza 128 segmenti
16-bit di spiazzamento indirizzano 65.536 byte

INDIRIZZO SEGMENTATO CORTO
15 bit indirizzano 32.768 byte
il numero del segmento a 7-bit indirizza 128 segmenti
8-bit di spiazzamento indirizzano 256 byte

Formati degli Indirizzi Usati nelle Istruzioni

APPENDICE D

Formato dei Registri di Controllo

Identificazione del significato dei Bit nella Parola degli Indicatori di Controllo (FCW)

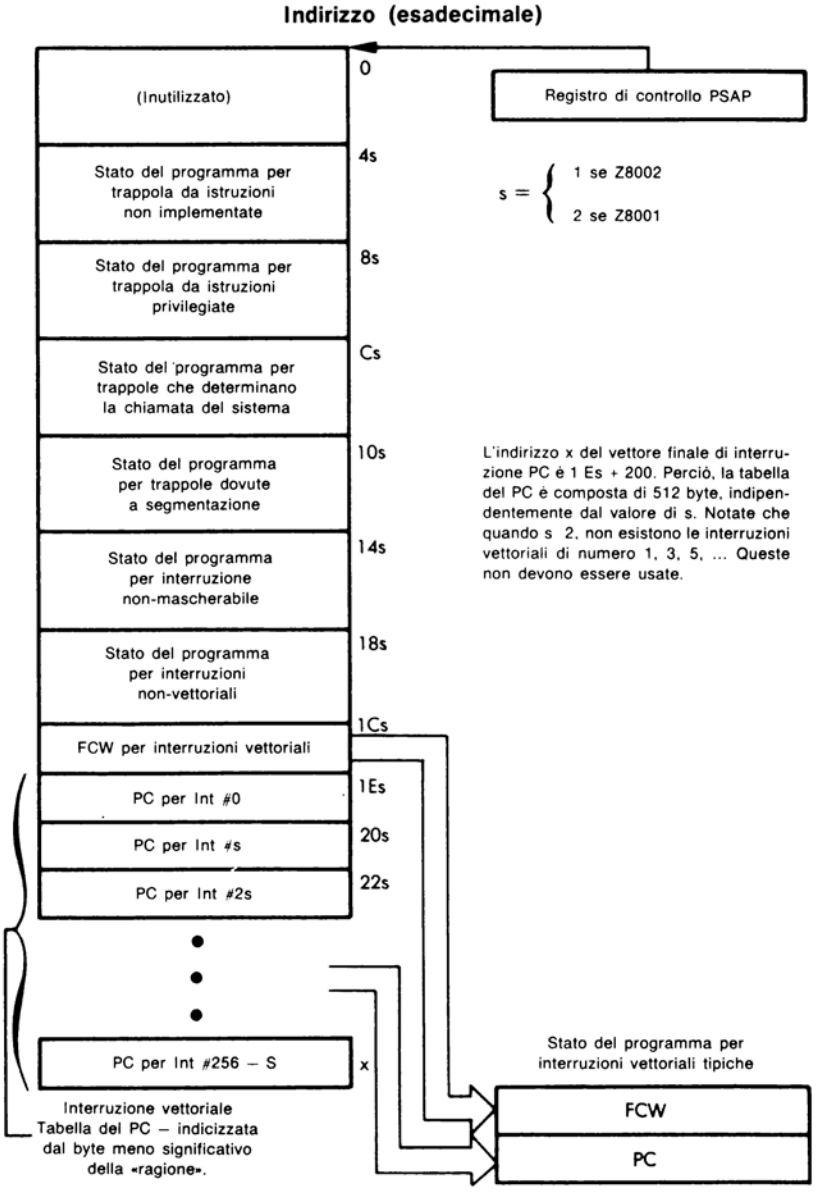
Localizzazione dei Campi del Registro REFRESH

Gli 8 bit meno significativi dello spiazzamento di PSAP sono sempre a zero.

Formato di PSAP e NSP

APPENDICE E

Struttura dell'Area di Stato del Programma



Stato del Programma per le Interruzioni e le Trappole

L. 22.000

Cod. 321 D

L'AUTORE

L'esperienza del Dr. S. Richard Mateosian nella progettazione di piccoli computer risale alla prima apparizione dei microcomputers nel 1960. È stato manager di sviluppo sistemi per parecchie società dell'Area di S. Francisco, relatore al Medical Information Science all'U.C.S. S. Francisco e consulente di sistemi computer indipendenti.
Il Dr. Mateosian è laureato in matematica all'U.C. Berkeley.

GRUPPO
EDITORIALE
JACKSON

RICHARD
MATEOSIAN

La programmazione dello Z-800

32